

Design of Digital Circuits

Lecture 13: Pipelining

Prof. Onur Mutlu

ETH Zurich

Spring 2019

4 April 2019

Required Readings

■ This week

- Pipelining
 - H&H, Chapter 7.5
- Pipelining Issues
 - H&H, Chapter 7.8.1-7.8.3

■ Next week

- Out-of-order execution
 - H&H, Chapter 7.8-7.9
- Smith and Sohi, “The Microarchitecture of Superscalar Processors,” Proceedings of the IEEE, 1995
 - More advanced pipelining
 - Interrupt and exception handling
 - Out-of-order and superscalar execution concepts

Agenda for Today & Next Few Lectures

- Last week
 - Single-cycle Microarchitectures
 - Multi-cycle Microarchitectures

- This week
 - Pipelining
 - Issues in Pipelining: Control & Data Dependence Handling, State Maintenance and Recovery, ...

- Next week
 - Out-of-Order Execution
 - Issues in OoO Execution: Load-Store Handling, ...

Can We Do Better?

Can We Do Better?

- What limitations do you see with the multi-cycle design?
- Limited concurrency
 - Some hardware resources are idle during different phases of instruction processing cycle
 - “Fetch” logic is idle when an instruction is being “decoded” or “executed”
 - Most of the datapath is idle when a memory access is happening

Can We Use the Idle Hardware to Improve Concurrency?

- Goal: **More concurrency → Higher instruction throughput** (i.e., more “work” completed in one cycle)
- Idea: When an instruction is using some resources in its processing phase, **process other instructions on idle resources** not needed by that instruction
 - E.g., when an instruction is being decoded, fetch the next instruction
 - E.g., when an instruction is being executed, decode another instruction
 - E.g., when an instruction is accessing data memory (ld/st), execute the next instruction
 - E.g., when an instruction is writing its result into the register file, access data memory for the next instruction

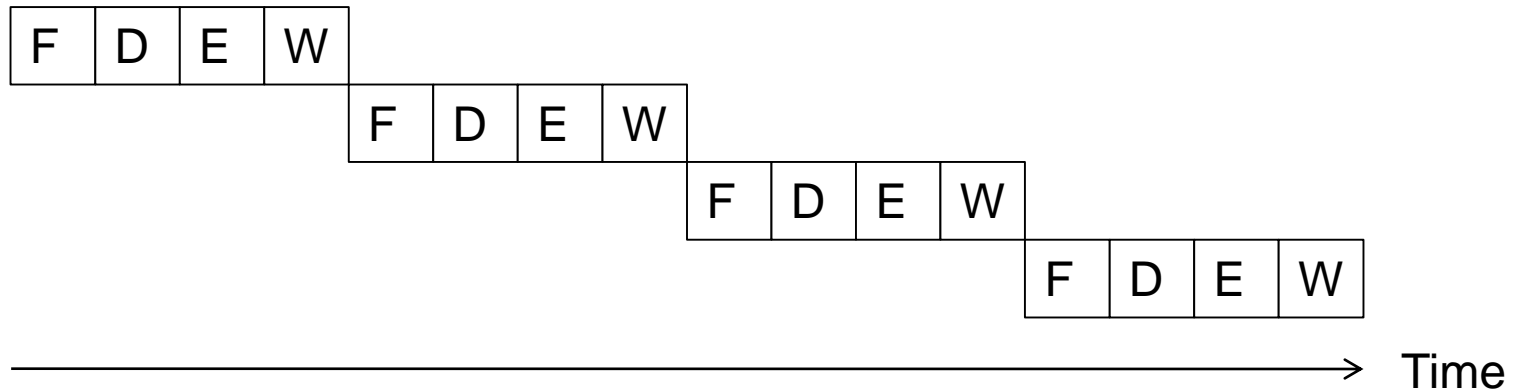
Pipelining

Pipelining: Basic Idea

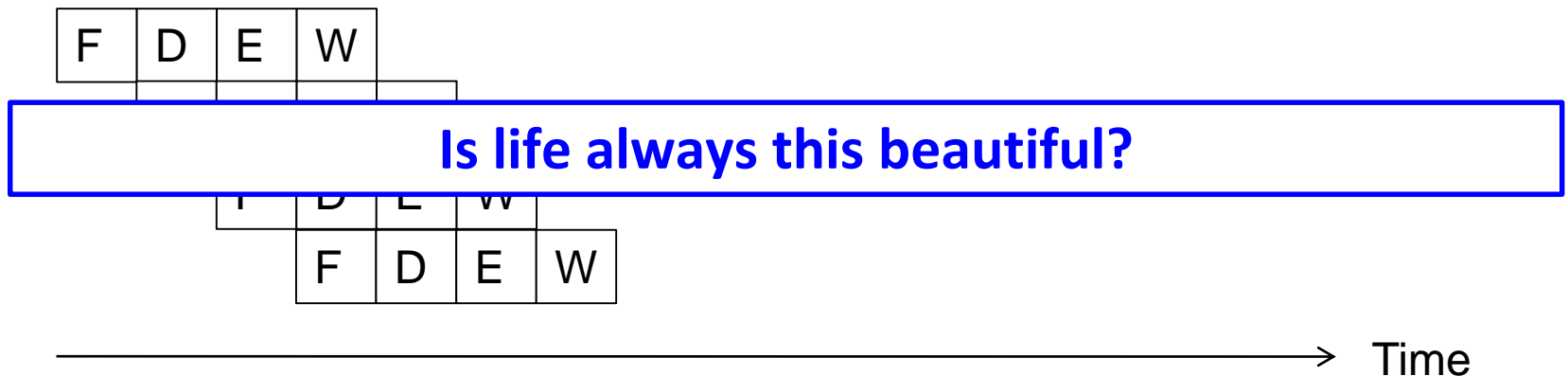
- More systematically:
 - Pipeline the execution of multiple instructions
 - Analogy: “Assembly line processing” of instructions
- Idea:
 - Divide the instruction processing cycle into distinct “stages” of processing
 - Ensure there are enough hardware resources to process one instruction in each stage
 - Process a **different** instruction in each stage
 - Instructions consecutive in program order are processed in consecutive stages
- Benefit: Increases instruction processing throughput (1/CPI)
- Downside: Start thinking about this...

Example: Execution of Four Independent ADDs

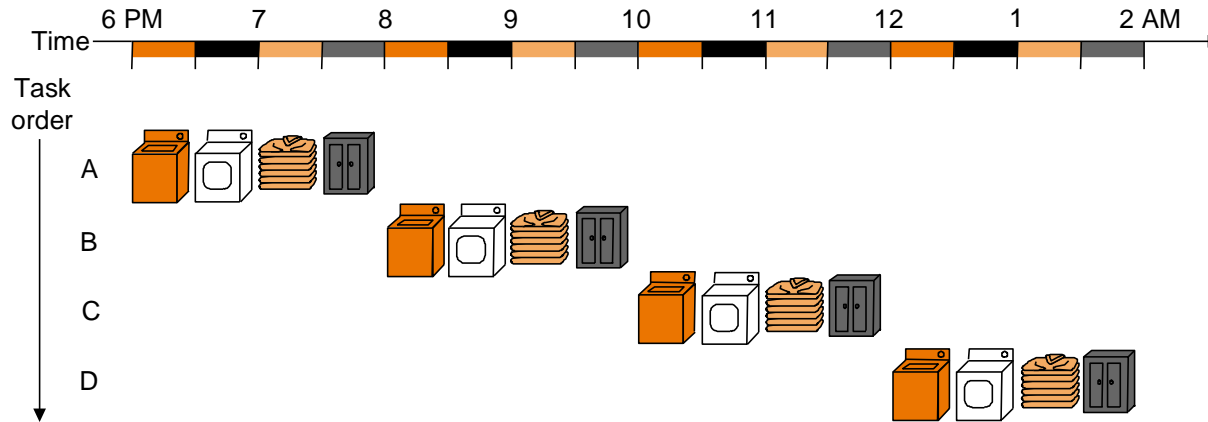
- Multi-cycle: 4 cycles per instruction



- Pipelined: 4 cycles per 4 instructions (steady state)

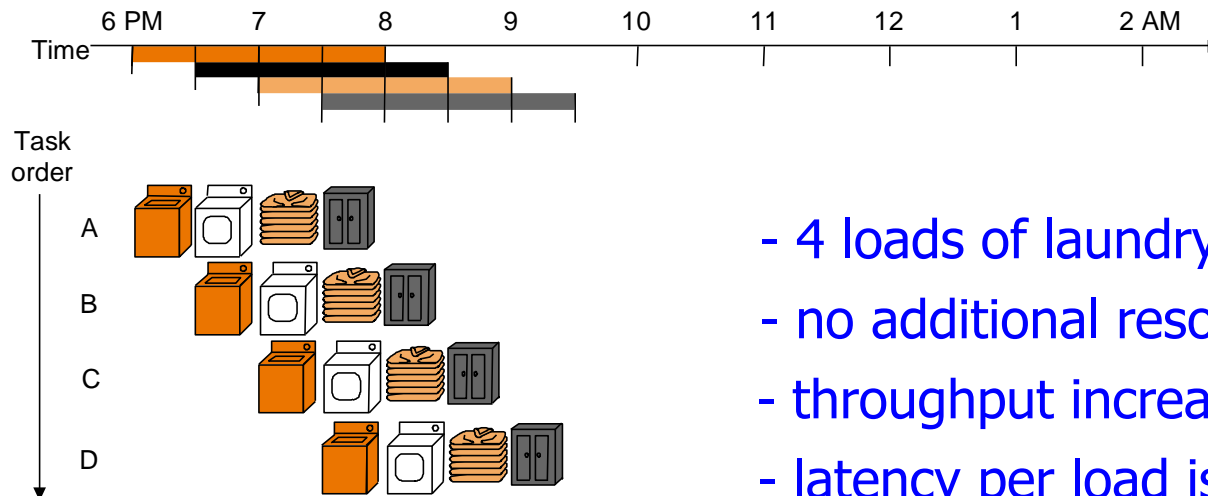
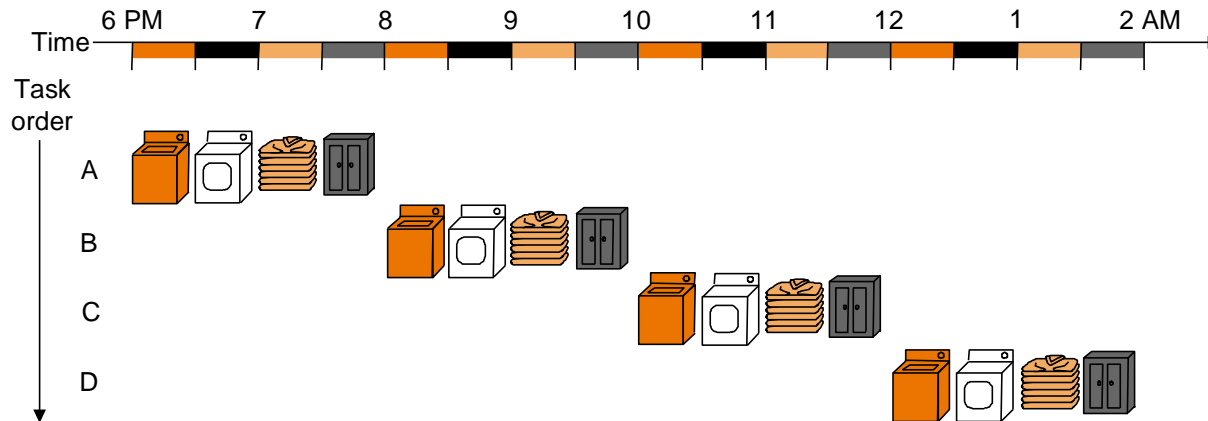


The Laundry Analogy



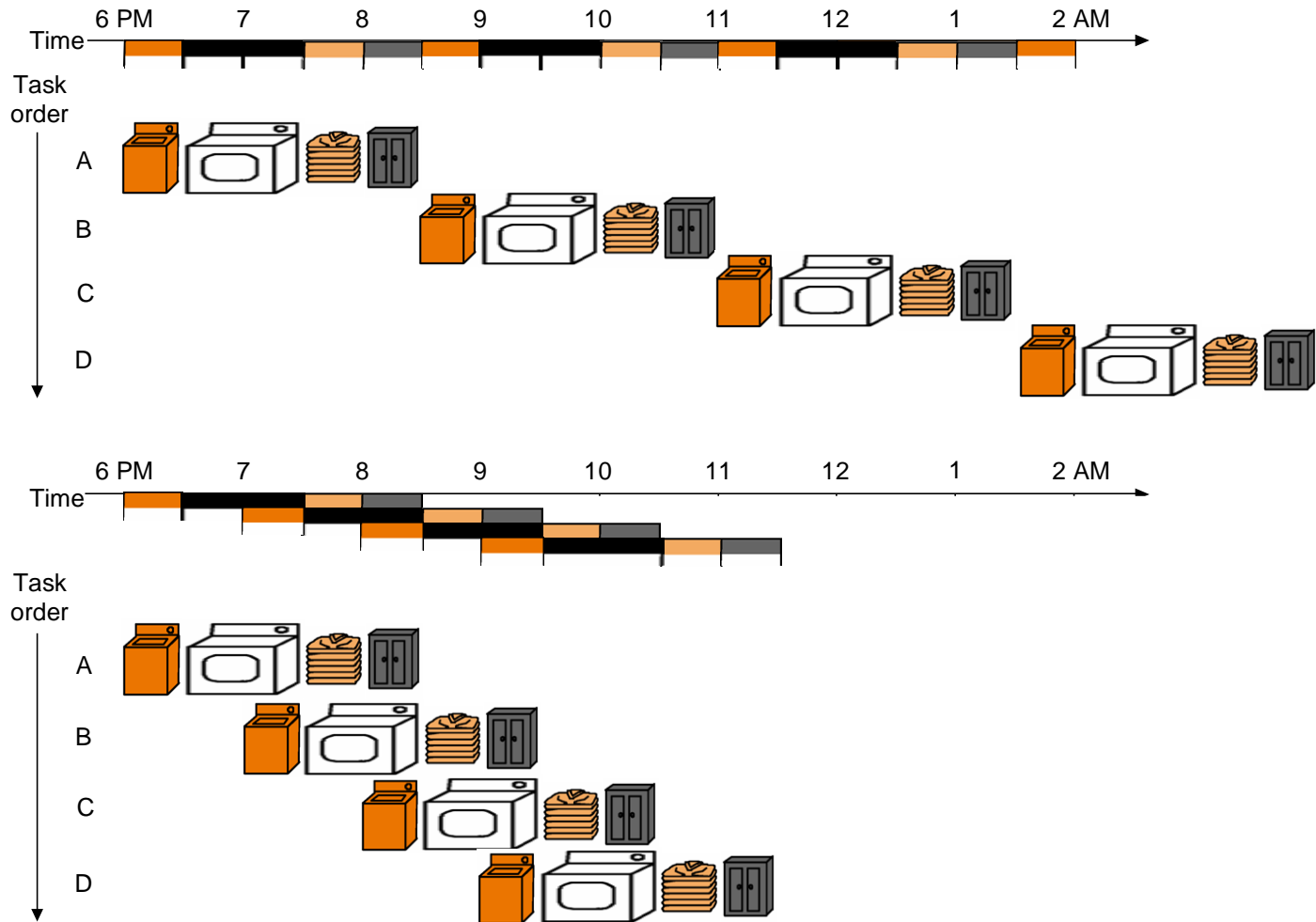
- “place one dirty load of clothes in the washer”
 - “when the washer is finished, place the wet load in the dryer”
 - “when the dryer is finished, take out the dry load and fold”
 - “when folding is finished, ask your roommate (??) to put the clothes away”
- steps to do a load are sequentially dependent
 - no dependence between different loads
 - different steps do not share resources

Pipelining Multiple Loads of Laundry



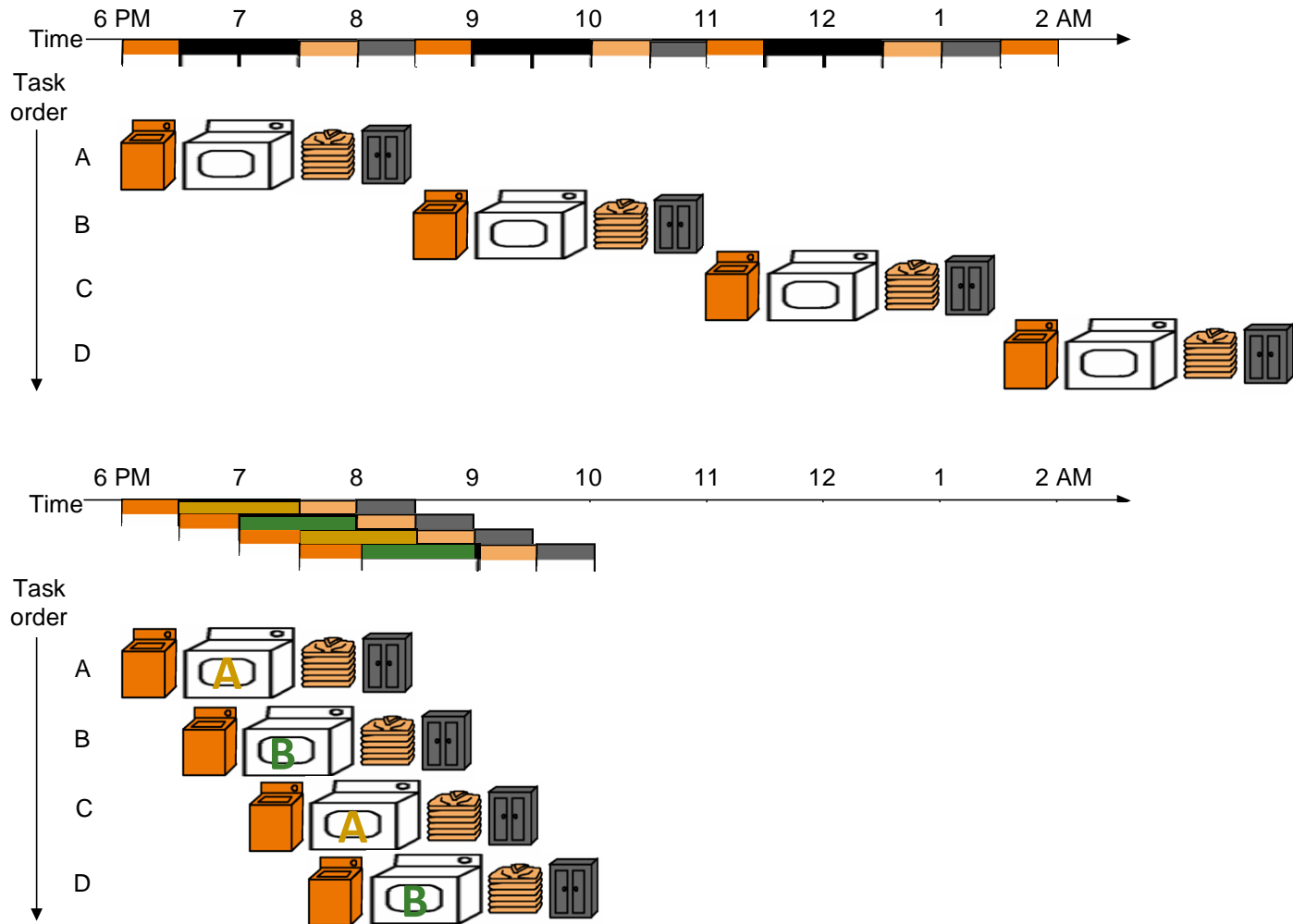
- 4 loads of laundry in parallel
- no additional resources
- throughput increased by 4
- latency per load is the same

Pipelining Multiple Loads of Laundry: In Practice



the slowest step decides throughput

Pipelining Multiple Loads of Laundry: In Practice

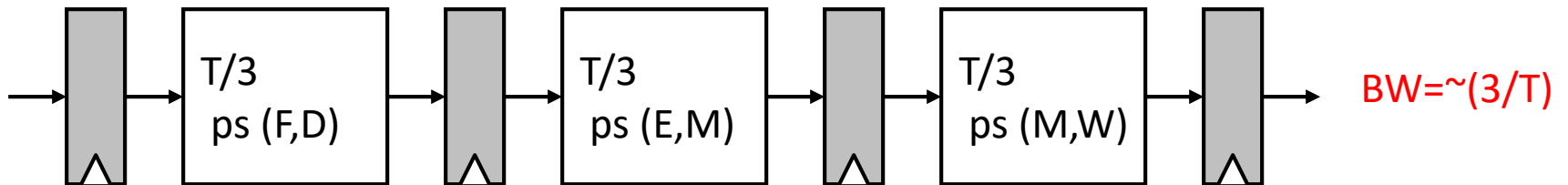
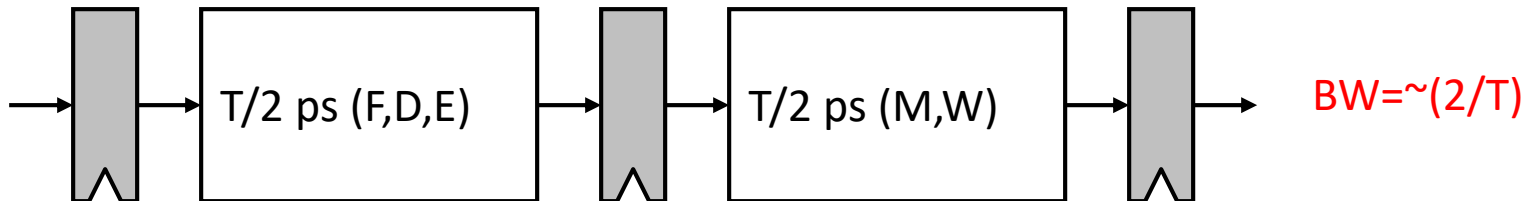


throughput restored (2 loads per hour) using 2 dryers

An Ideal Pipeline

- Goal: **Increase throughput with little increase in cost**
(hardware cost, in case of instruction processing)
- Repetition of **identical operations**
 - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of **independent operations**
 - No dependencies between repeated operations
- **Uniformly partitionable suboperations**
 - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
 - What about the instruction processing “cycle”?

Ideal Pipelining



More Realistic Pipeline: Throughput

- Nonpipelined version with delay T

$$BW = 1/(T+S) \text{ where } S = \text{latch delay}$$

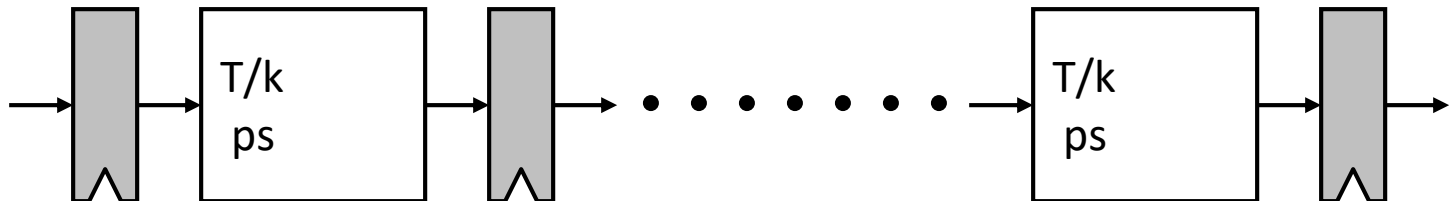


- k-stage pipelined version

$$BW_{k\text{-stage}} = 1 / (T/k + S)$$

$$BW_{\max} = 1 / (1 \text{ gate delay} + S)$$

**Latch delay reduces throughput
(switching overhead b/w stages)**



More Realistic Pipeline: Cost

- Nonpipelined version with combinational cost G

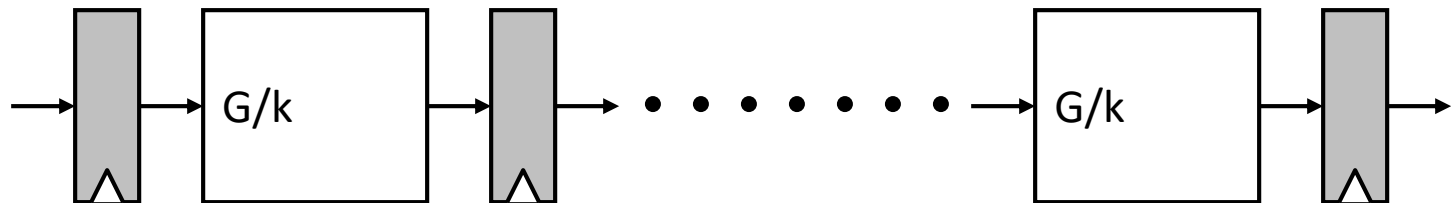
$\text{Cost} = G + L$ where L = latch cost



- k -stage pipelined version

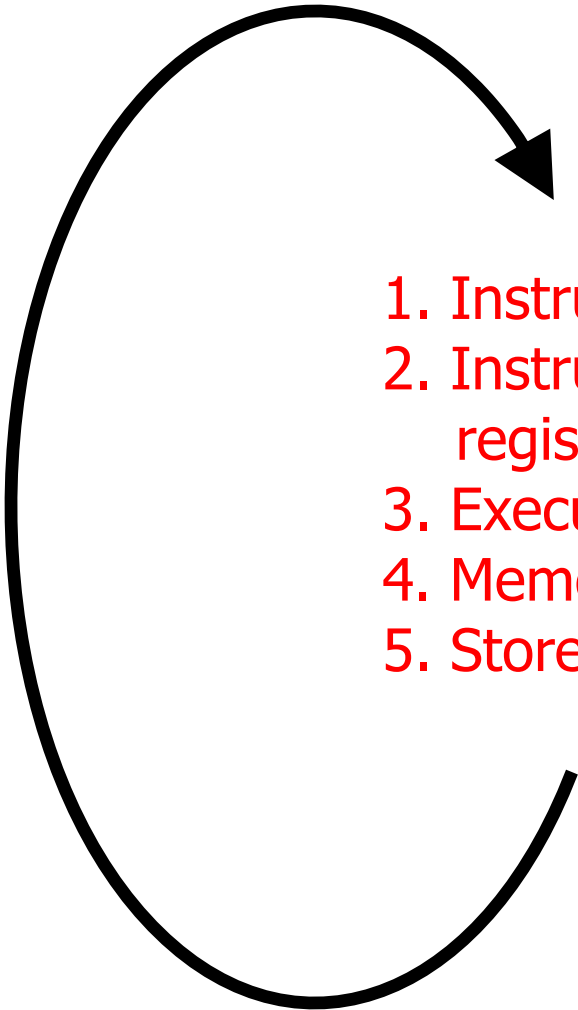
$\text{Cost}_{k\text{-stage}} = G + Lk$

Latches increase hardware cost

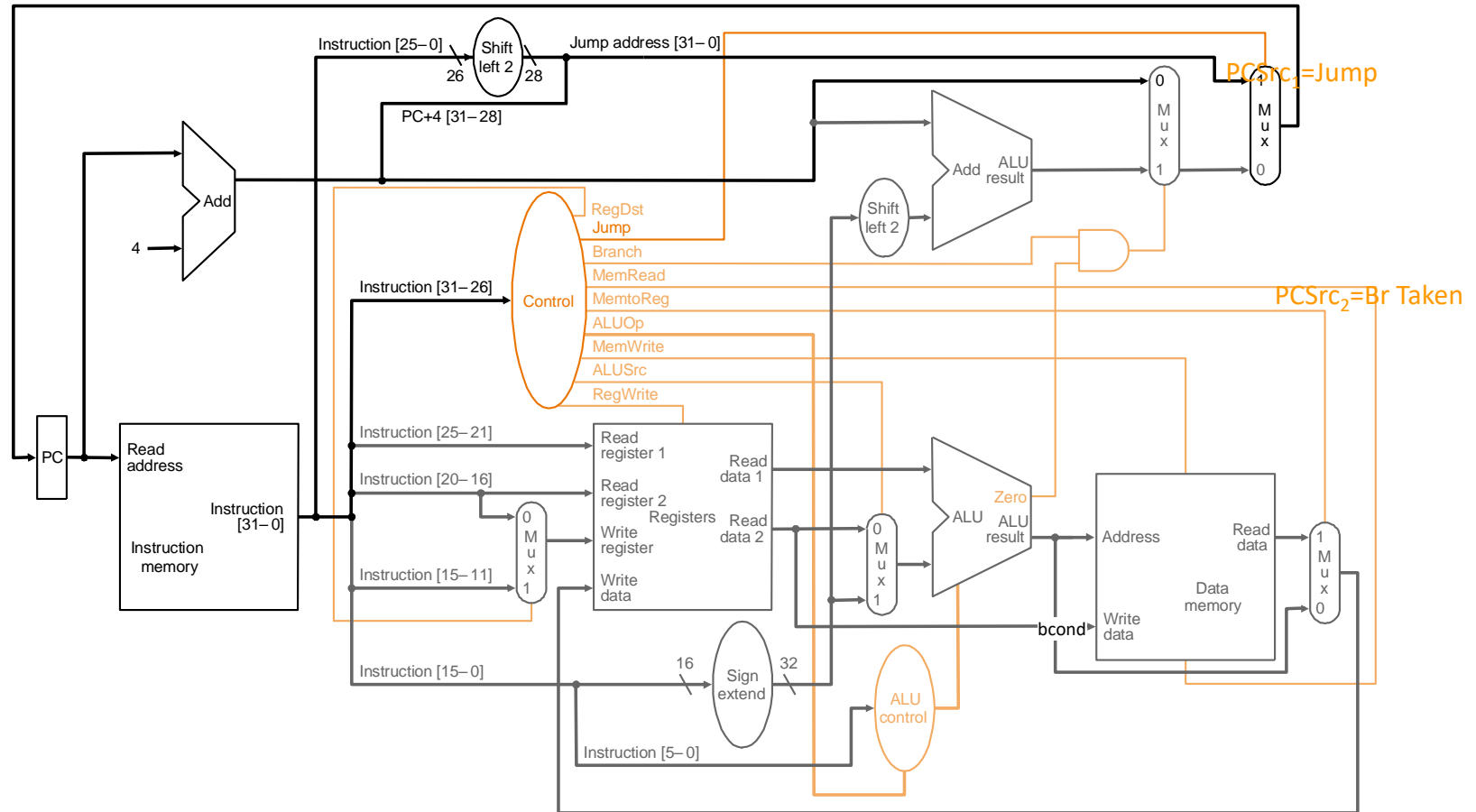


Pipelining Instruction Processing

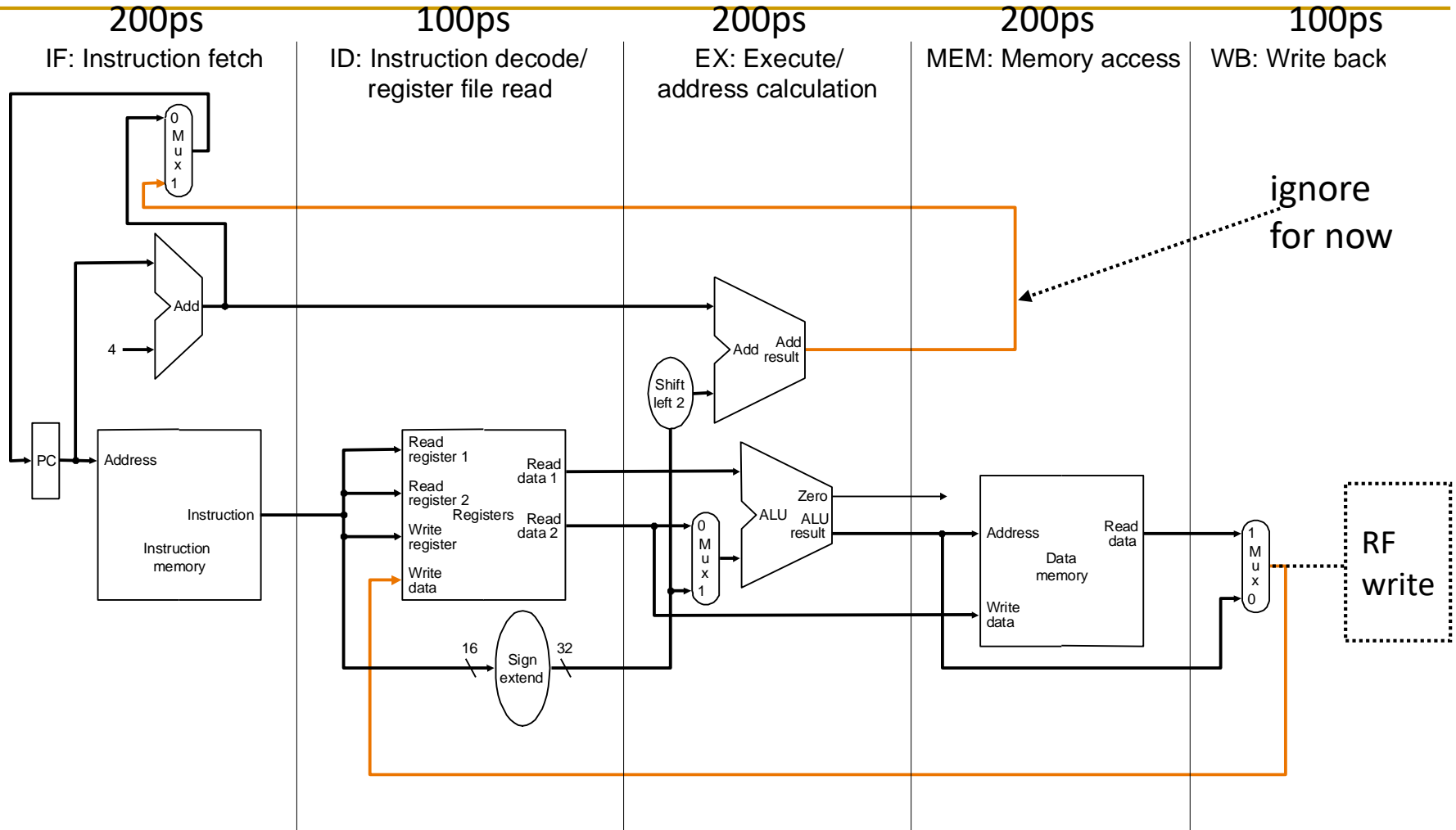
Remember: The Instruction Processing Cycle

- 
1. Instruction fetch (IF)
 2. Instruction decode and register operand fetch (ID/RF)
 3. Execute/Evaluate memory address (EX/AG)
 4. Memory operand fetch (MEM)
 5. Store/writeback result (WB)

Remember the Single-Cycle Uarch



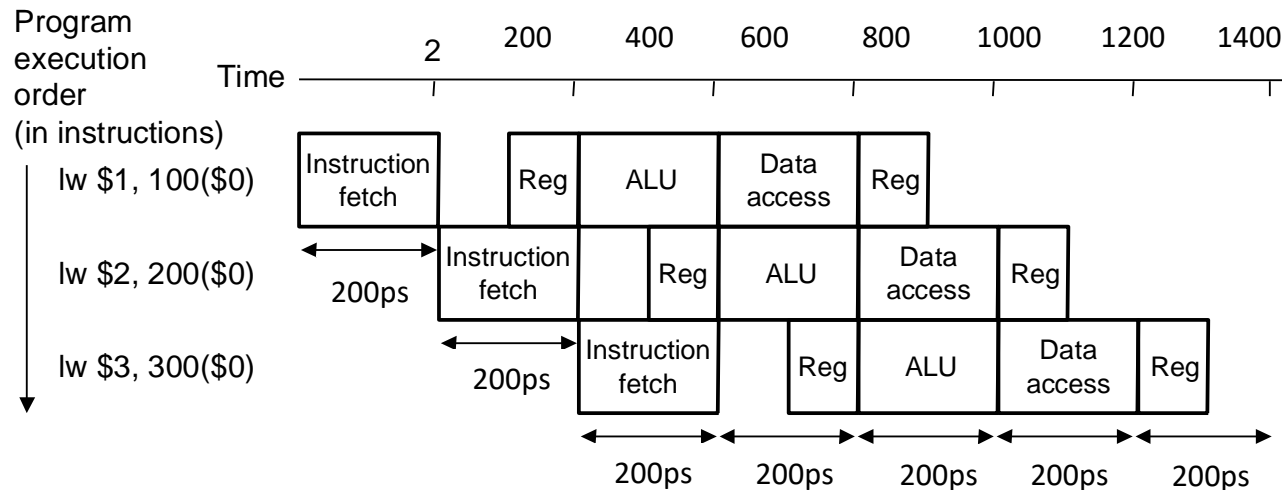
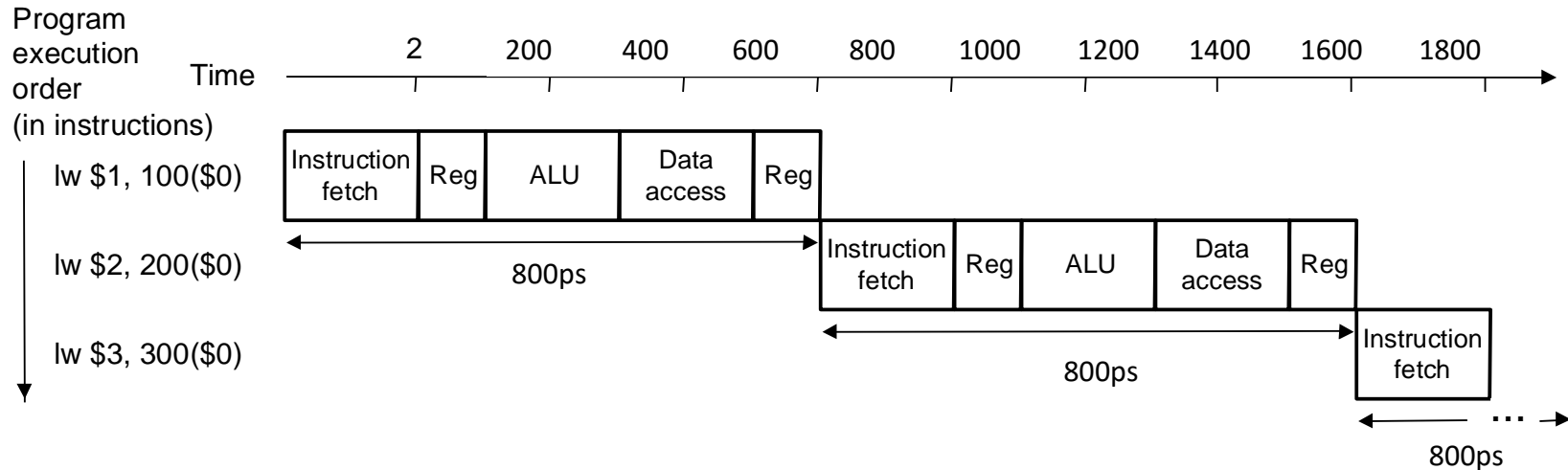
Dividing Into Stages



Is this the correct partitioning?

Why not 4 or 6 stages? Why not different boundaries?

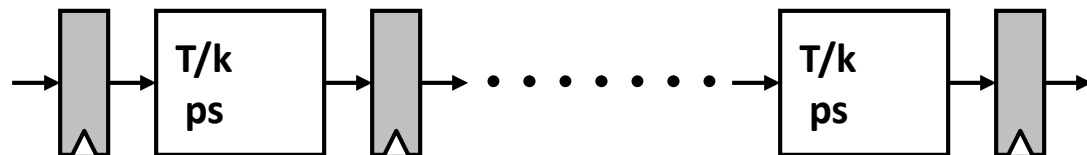
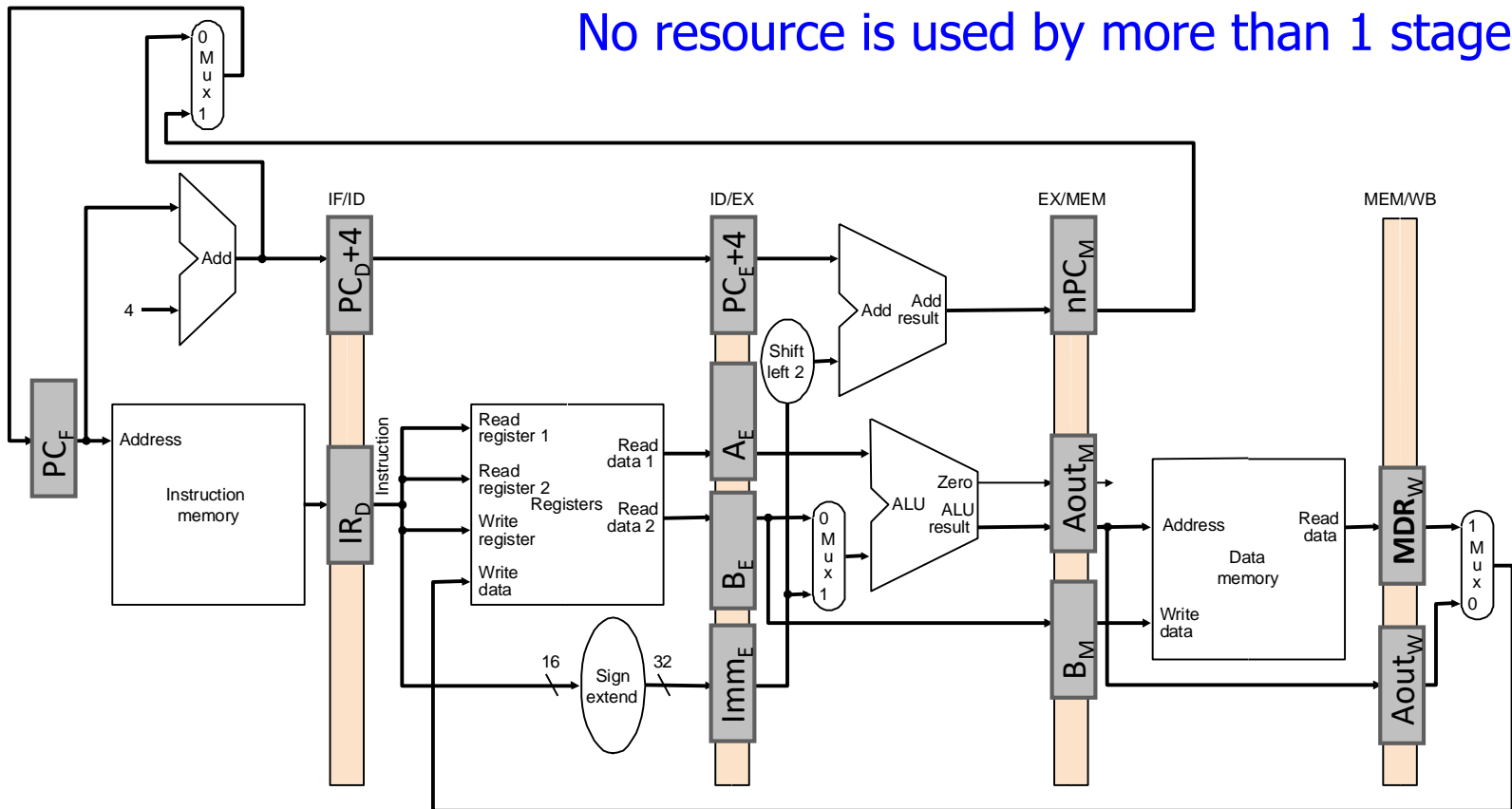
Instruction Pipeline Throughput



5-stage speedup is 4, not 5 as predicted by the ideal model. Why?

Enabling Pipelined Processing: Pipeline Registers

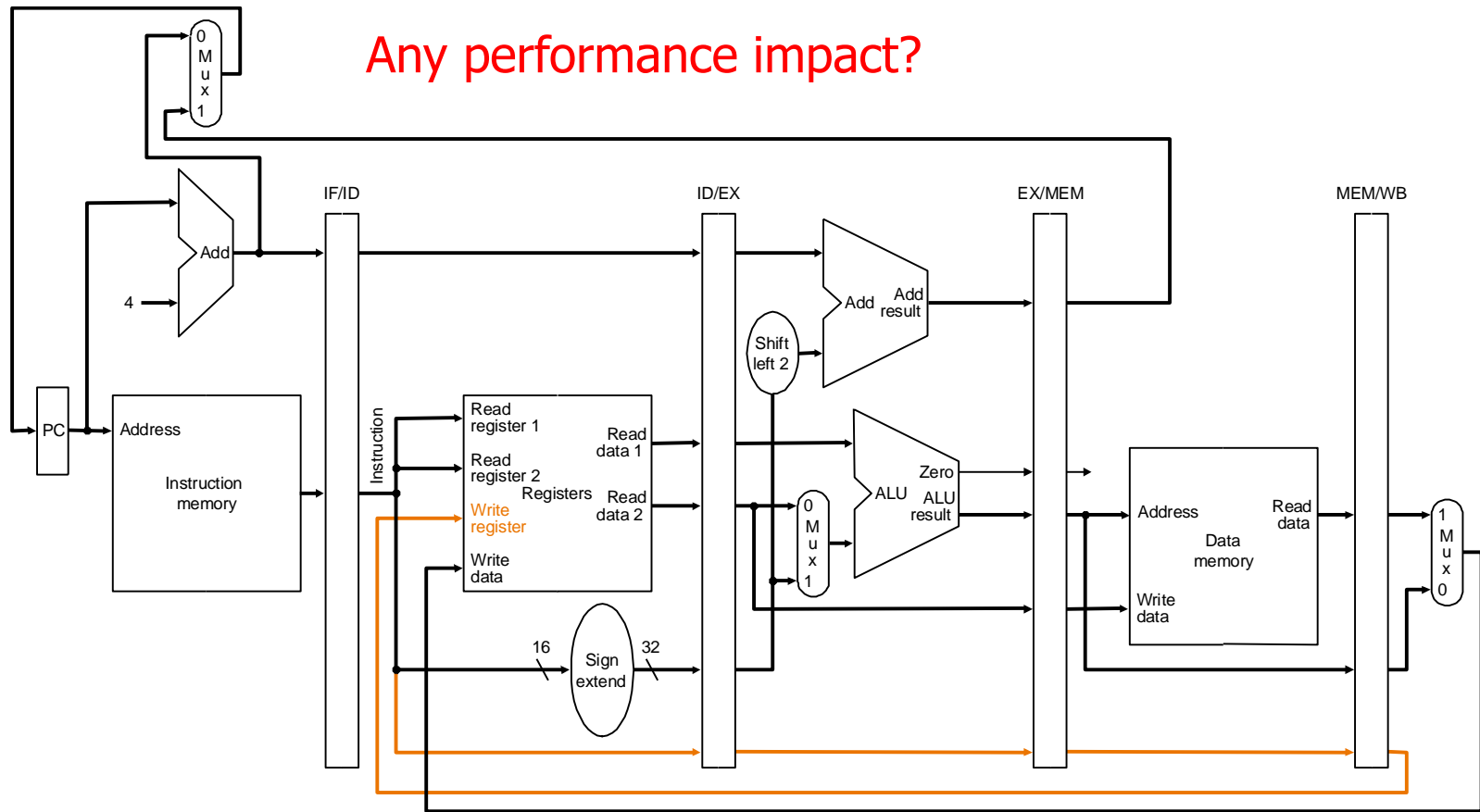
No resource is used by more than 1 stage!



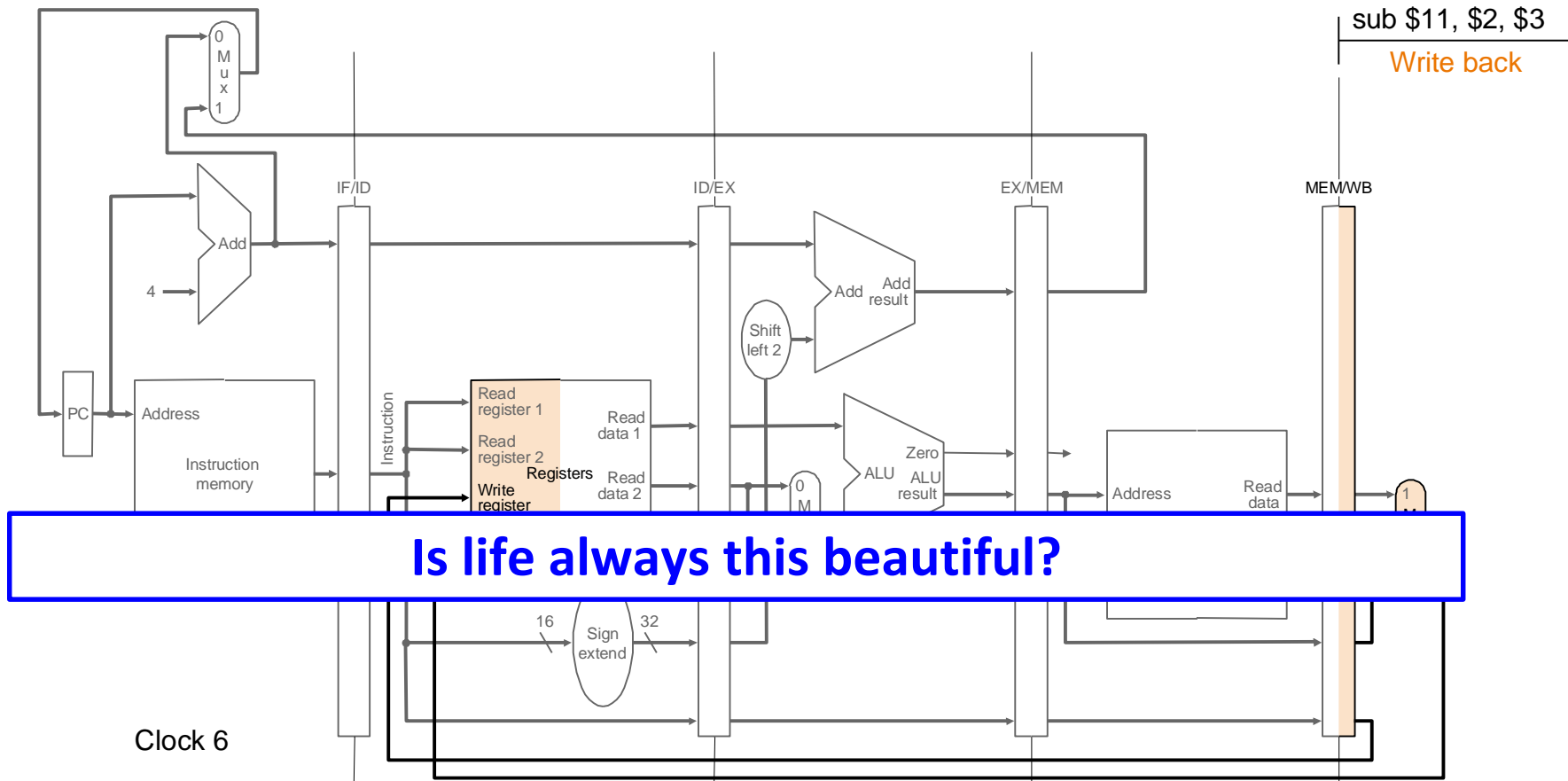
Pipelined Operation Example

All instruction classes must follow the same path and timing through the pipeline stages.

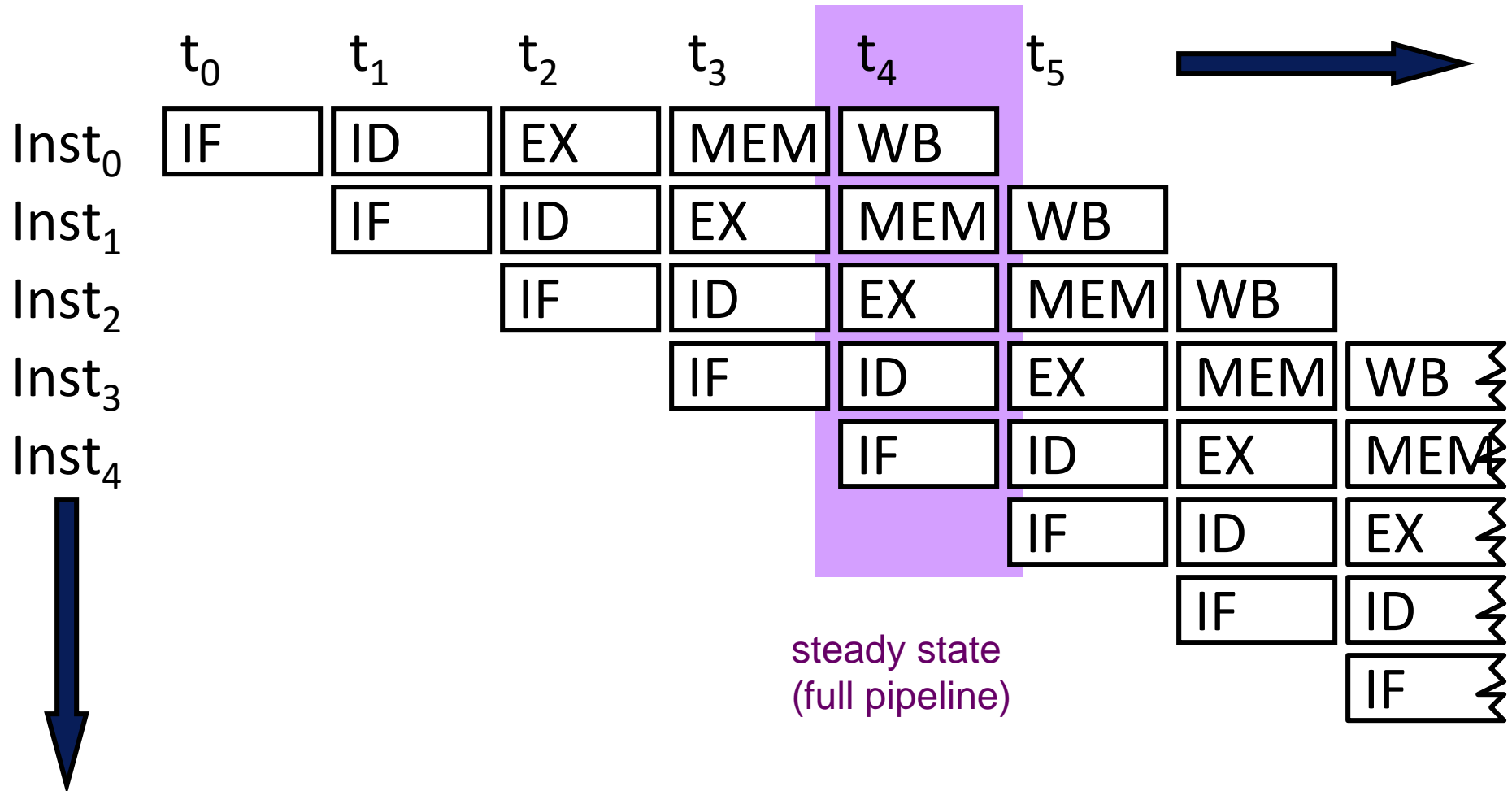
Any performance impact?



Pipelined Operation Example



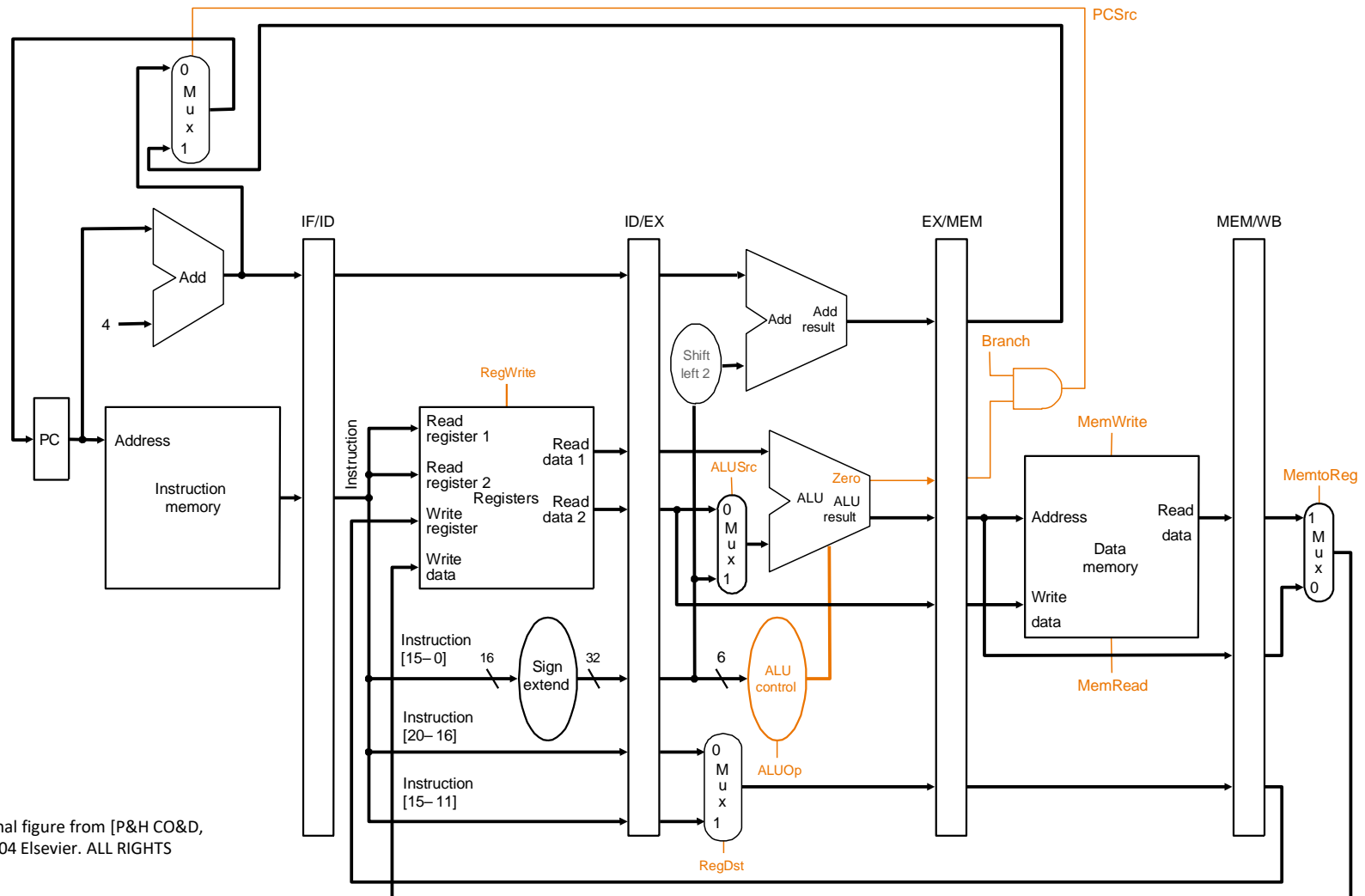
Illustrating Pipeline Operation: Operation View



Illustrating Pipeline Operation: Resource View

	t ₀	t ₁	t ₂	t ₃	t ₄	t ₅	t ₆	t ₇	t ₈	t ₉	t ₁₀
IF	I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈	I ₉	I ₁₀
ID		I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈	I ₉
EX			I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇	I ₈
MEM				I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆	I ₇
WB					I ₀	I ₁	I ₂	I ₃	I ₄	I ₅	I ₆

Control Points in a Pipeline



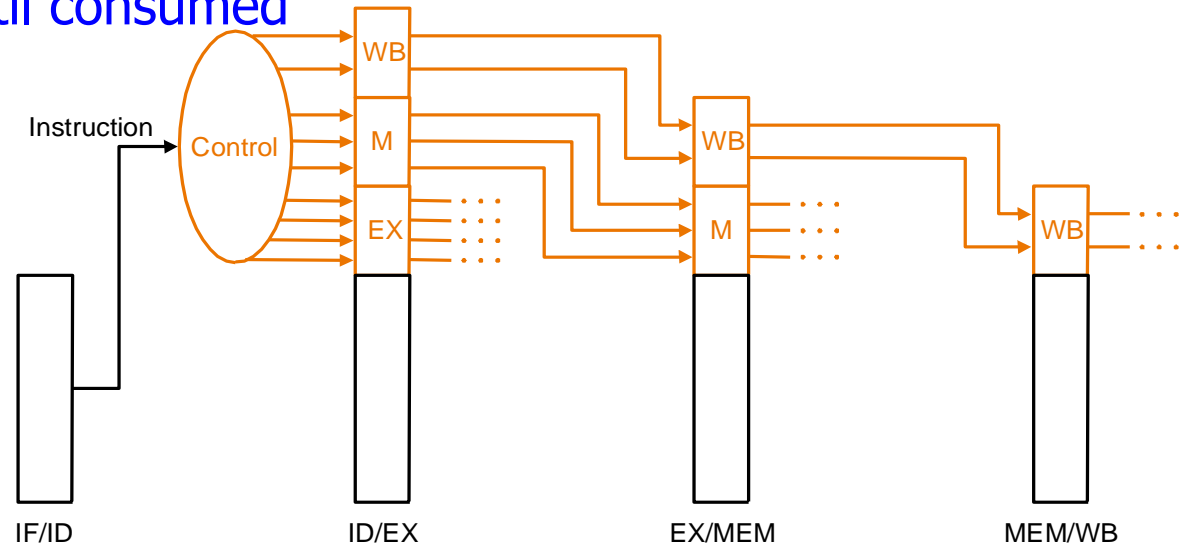
Based on original figure from [P&H CO&D,
COPYRIGHT 2004 Elsevier. ALL RIGHTS
RESERVED.]

Identical set of control points as the single-cycle datapath!!

Control Signals in a Pipeline

■ For a given instruction

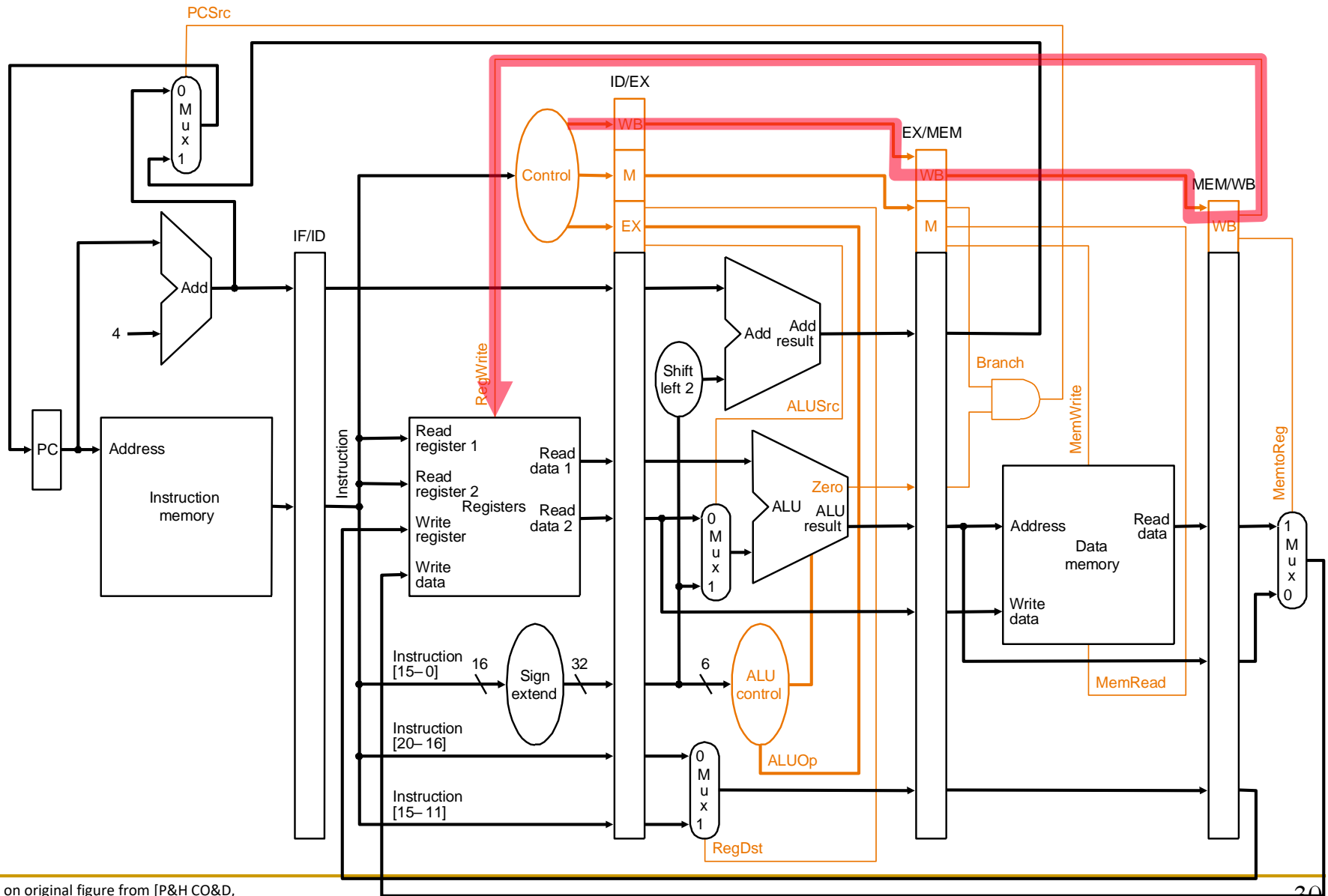
- same control signals as single-cycle, but
 - control signals required at different cycles, depending on stage
- ⇒ Option 1: decode once using the same logic as single-cycle and buffer signals until consumed



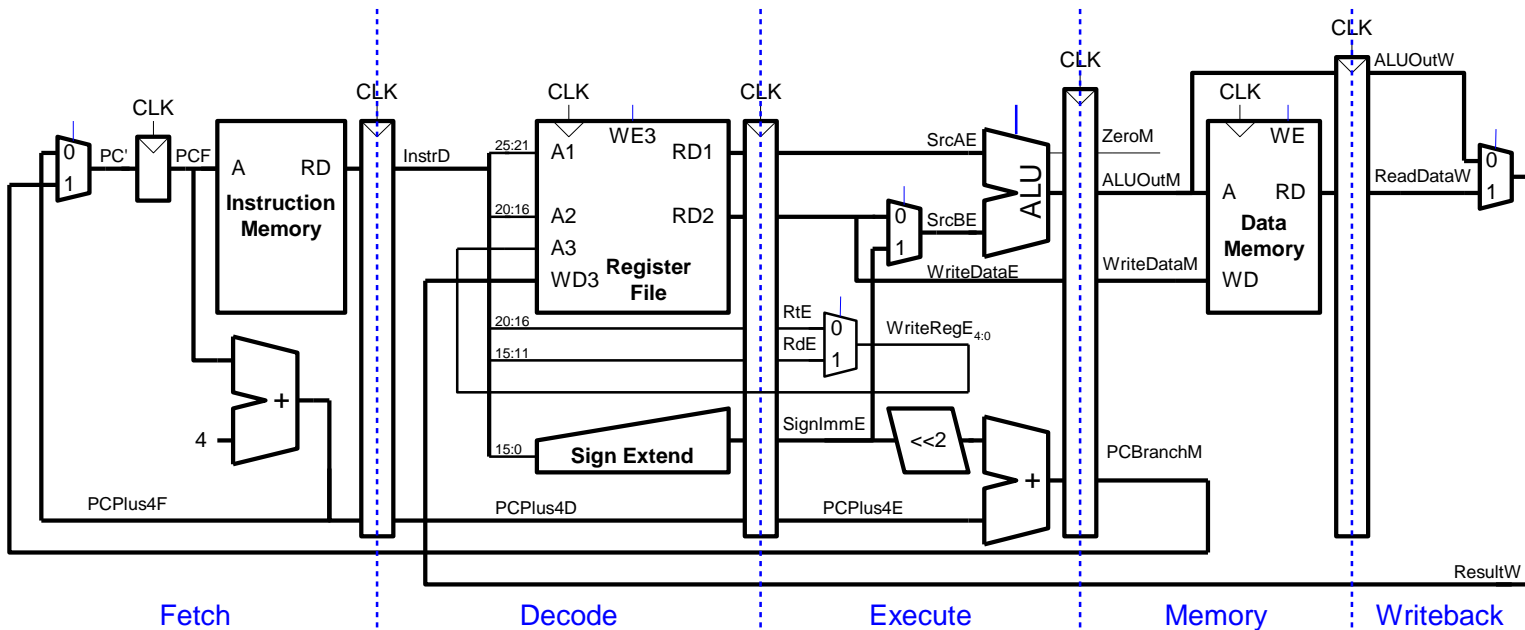
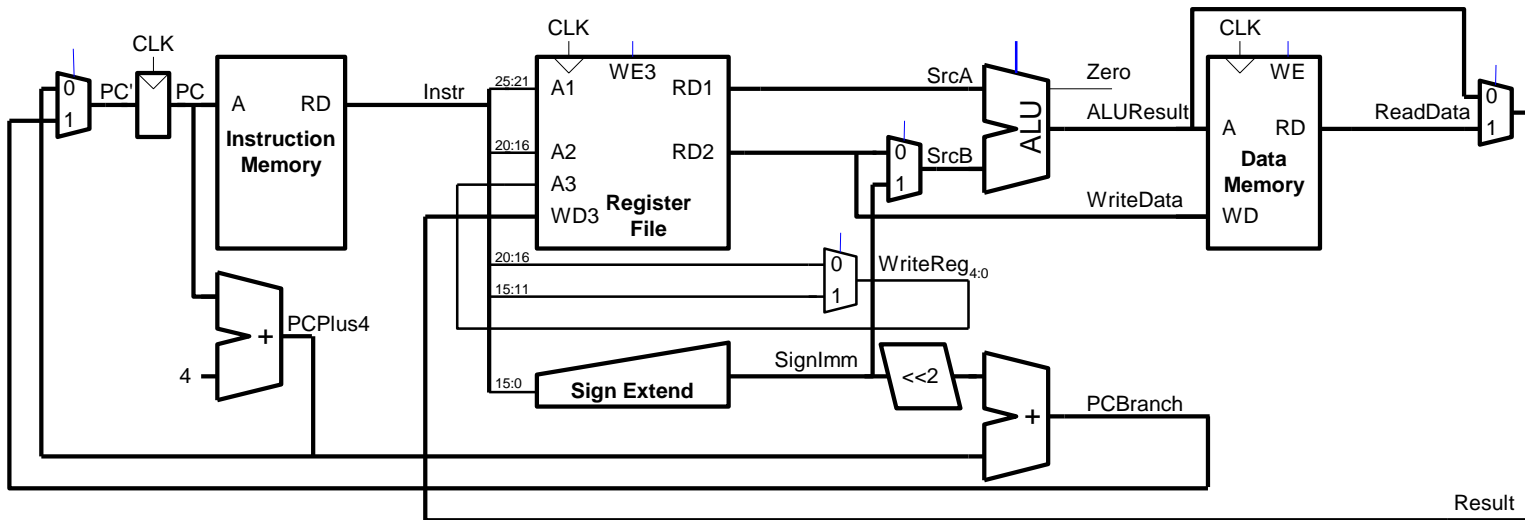
- ⇒ Option 2: carry relevant “instruction word/field” down the pipeline and decode locally within each or in a previous stage

Which one is better?

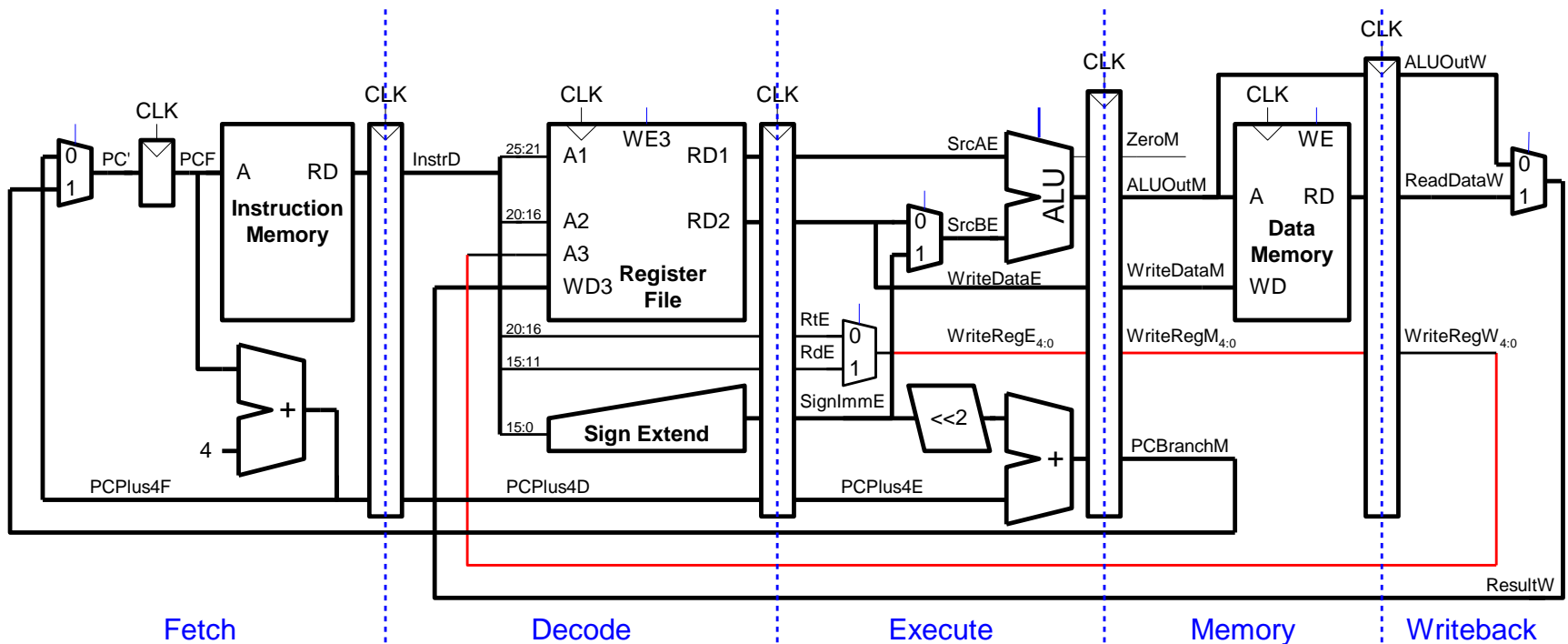
Pipelined Control Signals



Another Example: Single-Cycle and Pipelined

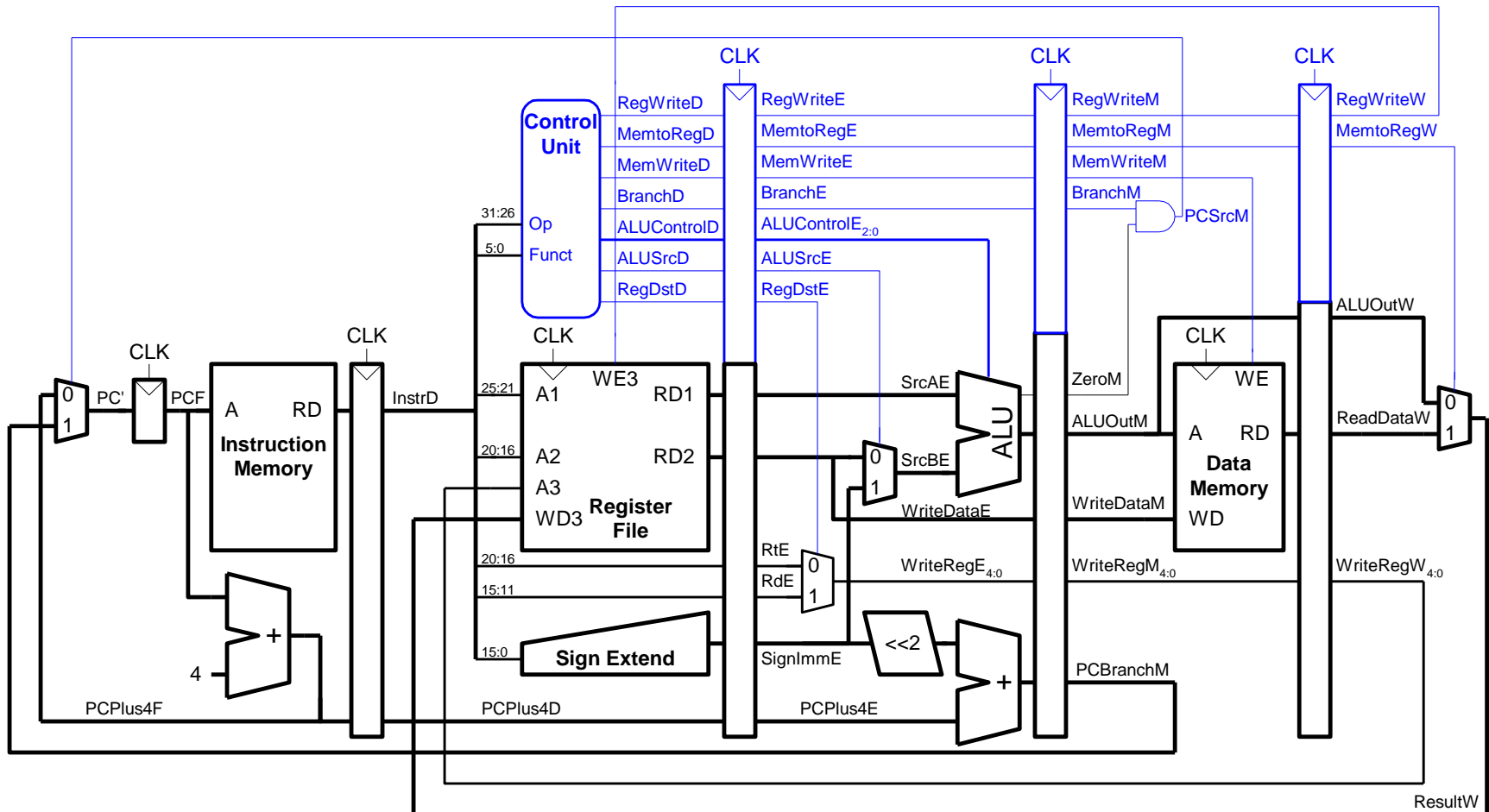


Another Example: Correct Pipelined Datapath



- WriteReg must arrive at the same time as Result

Another Example: Pipelined Control



- Same control unit as single-cycle processor
- Control delayed to proper pipeline stage

Remember: An Ideal Pipeline

- Goal: **Increase throughput with little increase in cost** (hardware cost, in case of instruction processing)
- Repetition of **identical operations**
 - The same operation is repeated on a large number of different inputs (e.g., all laundry loads go through the same steps)
- Repetition of **independent operations**
 - No dependencies between repeated operations
- **Uniformly partitionable suboperations**
 - Processing can be evenly divided into uniform-latency suboperations (that do not share resources)
- Fitting examples: automobile assembly line, doing laundry
 - **What about the instruction processing “cycle”?**

Instruction Pipeline: Not An Ideal Pipeline

- Identical operations ... NOT!

- ⇒ different instructions → not all need the same stages

- Forcing different instructions to go through the same pipe stages

- external fragmentation (some pipe stages idle for some instructions)

- Uniform suboperations ... NOT!

- ⇒ different pipeline stages → not the same latency

- Need to force each stage to be controlled by the same clock

- internal fragmentation (some pipe stages are too fast but all take the same clock cycle time)

- Independent operations ... NOT!

- ⇒ instructions are not independent of each other

- Need to detect and resolve inter-instruction dependencies to ensure the pipeline provides correct results

- pipeline stalls (pipeline is not always moving)

Issues in Pipeline Design

- Balancing work in pipeline stages
 - How many stages and what is done in each stage
- Keeping the pipeline correct, moving, and full in the presence of events that disrupt pipeline flow
 - Handling dependences
 - Data
 - Control
 - Handling resource contention
 - Handling long-latency (multi-cycle) operations
- Handling exceptions, interrupts
- Advanced: Improving pipeline throughput
 - Minimizing *stalls*

Causes of Pipeline *Stalls*

- Stall: A condition when the pipeline stops moving
- Resource contention
- Dependences (between instructions)
 - Data
 - Control
- Long-latency (multi-cycle) operations

Dependences and Their Types

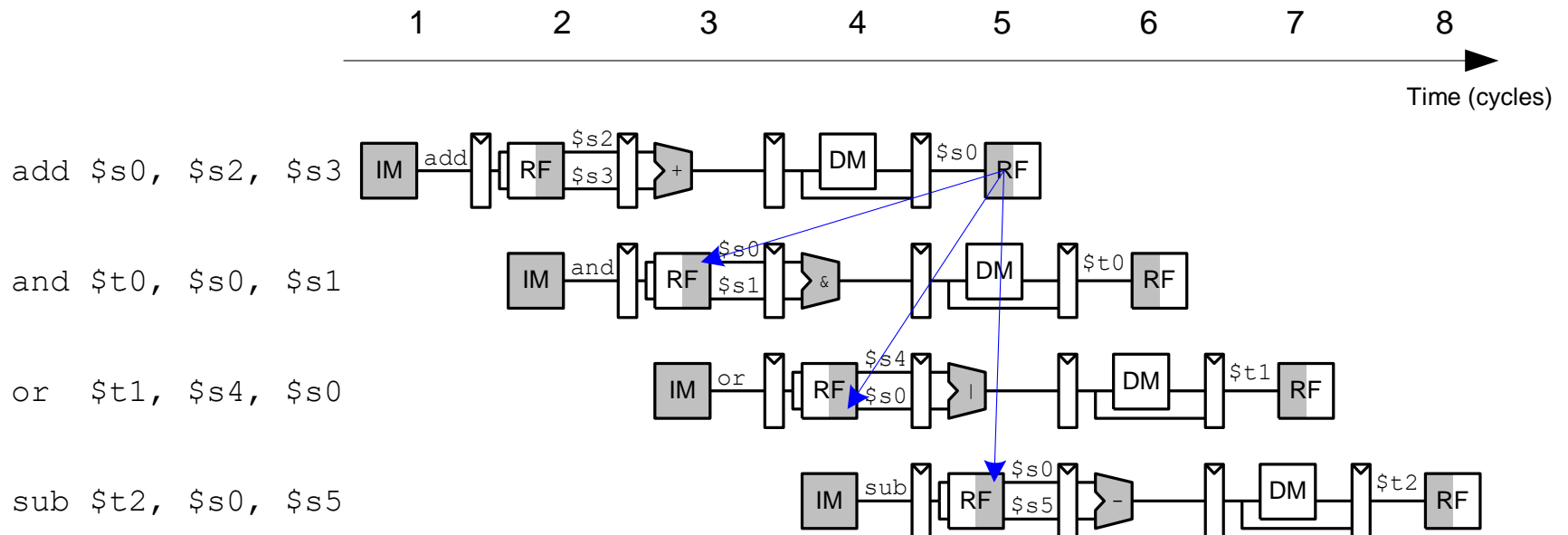
- Also called “dependency” or *less desirably* “hazard”
- Dependences dictate ordering requirements between instructions
- Two types
 - Data dependence
 - Control dependence
- Resource contention is sometimes called resource dependence
 - However, this is not fundamental to (dictated by) program semantics, so we will treat it separately

Handling Resource Contention

- Happens when instructions in two pipeline stages need the same resource
- Solution 1: Eliminate the cause of contention
 - Duplicate the resource or increase its throughput
 - E.g., use separate instruction and data memories (caches)
 - E.g., use multiple ports for memory structures
- Solution 2: Detect the resource contention and stall one of the contending stages
 - Which stage do you stall?
 - Example: What if you had a single read and write port for the register file?

Example Resource Dependence: RegFile

- The register file can be read and written in the same cycle:
 - write takes place during the 1st half of the cycle
 - read takes place during the 2nd half of the cycle => no problem!!!
 - However operations that involve register file have only *half a clock cycle* to complete the operation!!




Data Dependences

- Types of data dependences
 - Flow dependence (true data dependence – read after write)
 - Output dependence (write after write)
 - Anti dependence (write after read)
- Which ones cause stalls in a pipelined machine?
 - For all of them, we need to ensure semantics of the program is correct
 - Flow dependences always need to be obeyed because they constitute true dependence on a value
 - Anti and output dependences exist due to limited number of architectural registers
 - They are dependence on a name, not a value
 - We will later see what we can do about them

Data Dependence Types

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

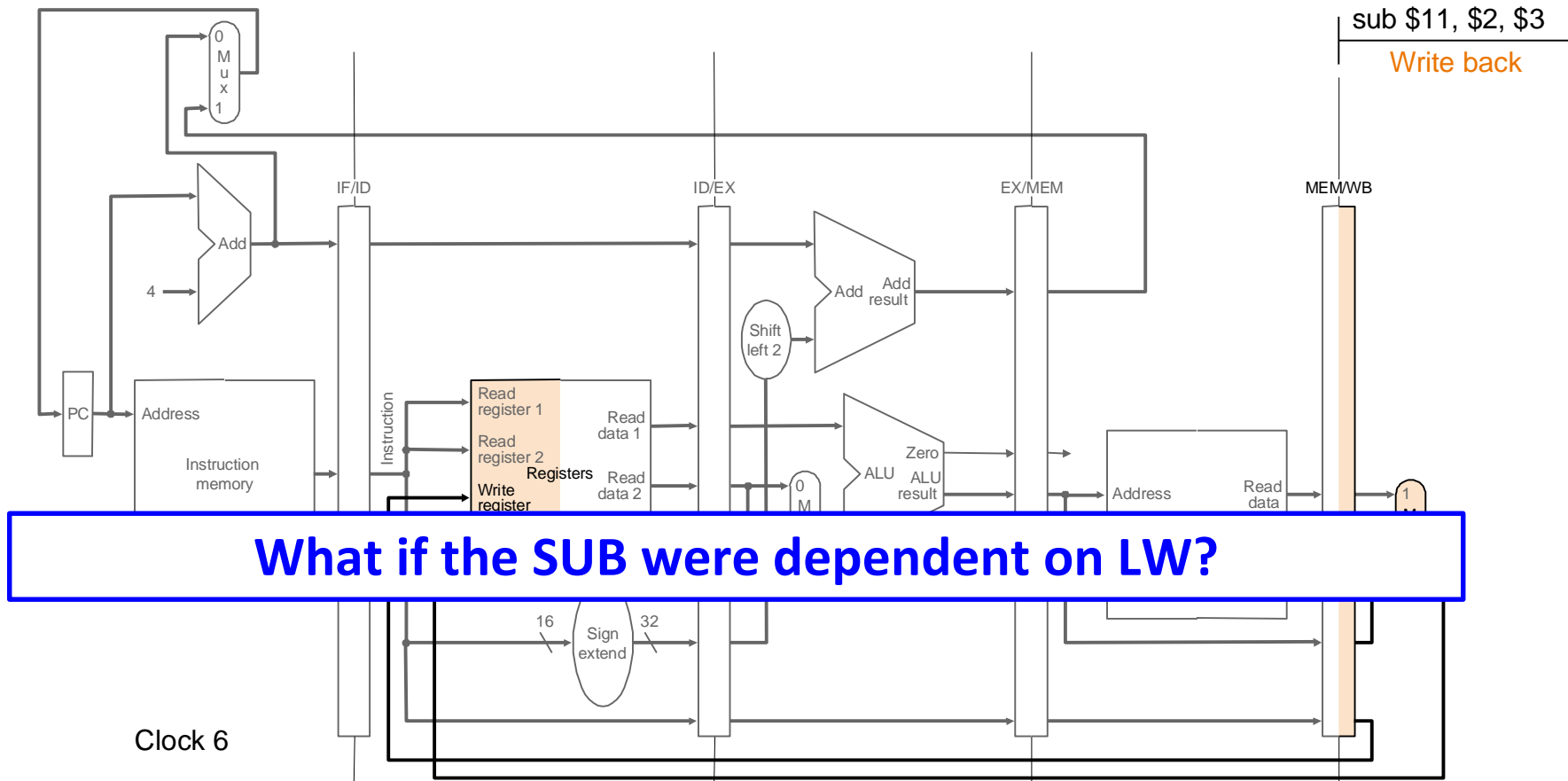
Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write
(WAW)

Pipelined Operation Example



Data Dependence Handling

Reading for Next Few Lectures

- H&H, Chapter 7.5-7.9
- Smith and Sohi, “[The Microarchitecture of Superscalar Processors](#),” Proceedings of the IEEE, 1995
 - More advanced pipelining
 - Interrupt and exception handling
 - Out-of-order and superscalar execution concepts

How to Handle Data Dependences

- Anti and output dependences are easier to handle
 - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
 - Detect and wait until value is available in register file
 - Detect and forward/bypass data to dependent instruction
 - Detect and eliminate the dependence at the software level
 - No need for the hardware to detect dependence
 - Predict the needed value(s), execute “speculatively”, and verify
 - Do something else (fine-grained multithreading)
 - No need to detect

Design of Digital Circuits

Lecture 13: Pipelining

Prof. Onur Mutlu

ETH Zurich

Spring 2019

4 April 2019

We did not cover the following slides.
They are for your preparation for the
next lecture.

Interlocking

- Detection of dependence between instructions in a pipelined processor to guarantee correct execution
- Software based interlocking
vs.
- Hardware based interlocking
- MIPS acronym?

Approaches to Dependence Detection (I)

■ Scoreboarding

- ❑ Each register in register file has a Valid bit associated with it
- ❑ An instruction that is writing to the register resets the Valid bit
- ❑ An instruction in Decode stage checks if all its source and destination registers are Valid
 - Yes: No need to stall... No dependence
 - No: Stall the instruction

■ Advantage:

- ❑ Simple. 1 bit per register

■ Disadvantage:

- ❑ Need to stall for all types of dependences, not only flow dep.

Not Stalling on Anti and Output Dependences

- What changes would you make to the scoreboard to enable this?

Approaches to Dependence Detection (II)

■ Combinational dependence check logic

- ❑ Special logic that checks if any instruction in later stages is supposed to write to any source register of the instruction that is being decoded
- ❑ Yes: stall the instruction/pipeline
- ❑ No: no need to stall... no flow dependence

■ Advantage:

- ❑ No need to stall on anti and output dependences

■ Disadvantage:

- ❑ Logic is more complex than a scoreboard
- ❑ Logic becomes more complex as we make the pipeline deeper and wider (flash-forward: think superscalar execution)

Once You Detect the Dependence in Hardware

- What do you do afterwards?
- Observation: Dependence between two instructions is detected before the communicated data value becomes available
- Option 1: Stall the dependent instruction right away
- Option 2: Stall the dependent instruction only when necessary → data forwarding/bypassing
- Option 3: ...

Data Forwarding/Bypassing

- Problem: A consumer (dependent) instruction has to wait in decode stage until the producer instruction writes its value in the register file
- Goal: We do not want to stall the pipeline unnecessarily
- Observation: The data value needed by the consumer instruction can be supplied directly from a later stage in the pipeline (instead of only from the register file)
- Idea: Add additional dependence check logic and data forwarding paths (buses) to supply the producer's value to the consumer right after the value is available
- Benefit: Consumer can move in the pipeline until the point the value can be supplied → less stalling

A Special Case of Data Dependence

- Control dependence
 - Data dependence on the Instruction Pointer / Program Counter

Control Dependence


- Question: **What should the fetch PC be in the next cycle?**
- Answer: The address of the next instruction
 - All instructions are control dependent on previous ones. Why?
- If the fetched instruction is a non-control-flow instruction:
 - Next Fetch PC is the address of the next-sequential instruction
 - Easy to determine if we know the size of the fetched instruction
- If the instruction that is fetched is a control-flow instruction:
 - How do we determine the next Fetch PC?
- In fact, how do we know whether or not the fetched instruction is a control-flow instruction?

Data Dependence Handling: Concepts and Implementation

Remember: Data Dependence Types

Flow dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$



Read-after-Write
(RAW)

Anti dependence


$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_1 \leftarrow r_4 \text{ op } r_5$



Write-after-Read
(WAR)

Output-dependence

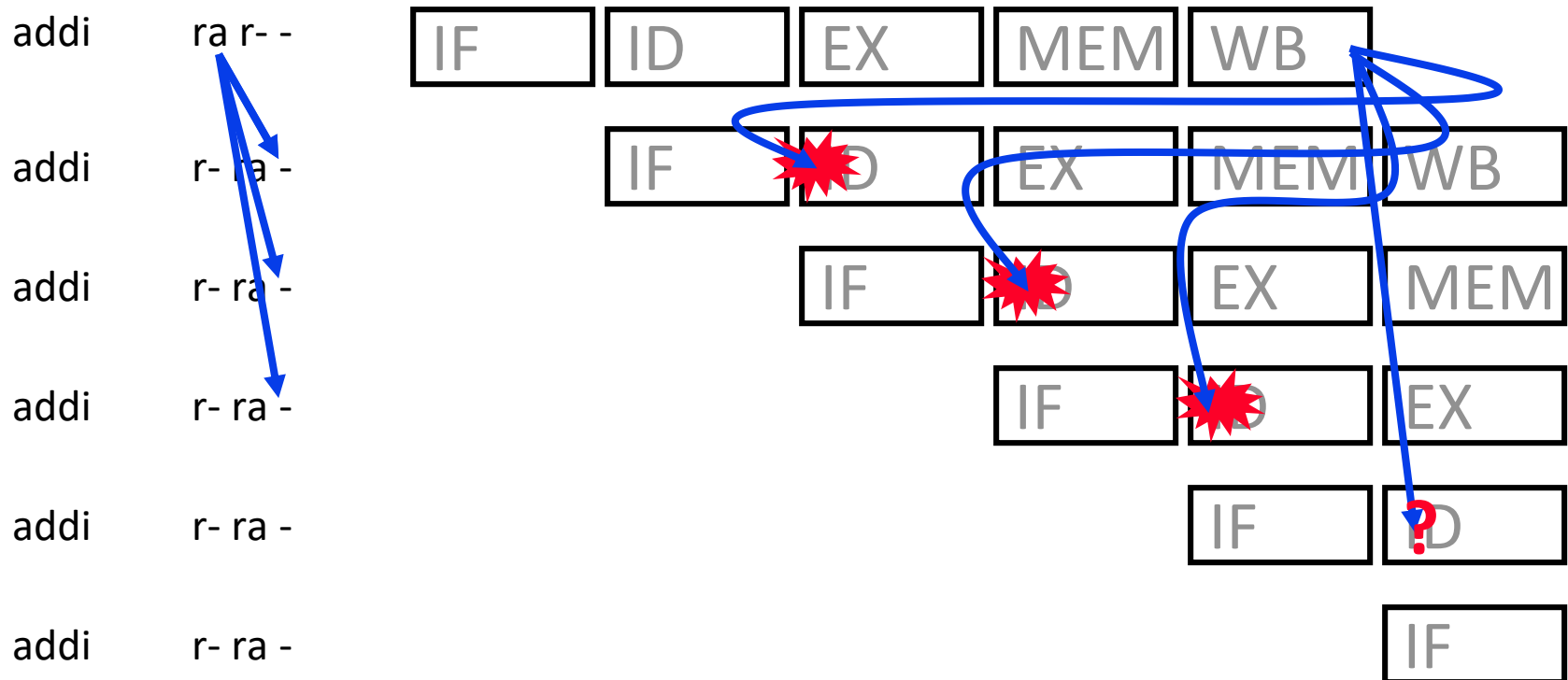
$r_3 \leftarrow r_1 \text{ op } r_2$
 $r_5 \leftarrow r_3 \text{ op } r_4$
 $r_3 \leftarrow r_6 \text{ op } r_7$



Write-after-Write
(WAW)

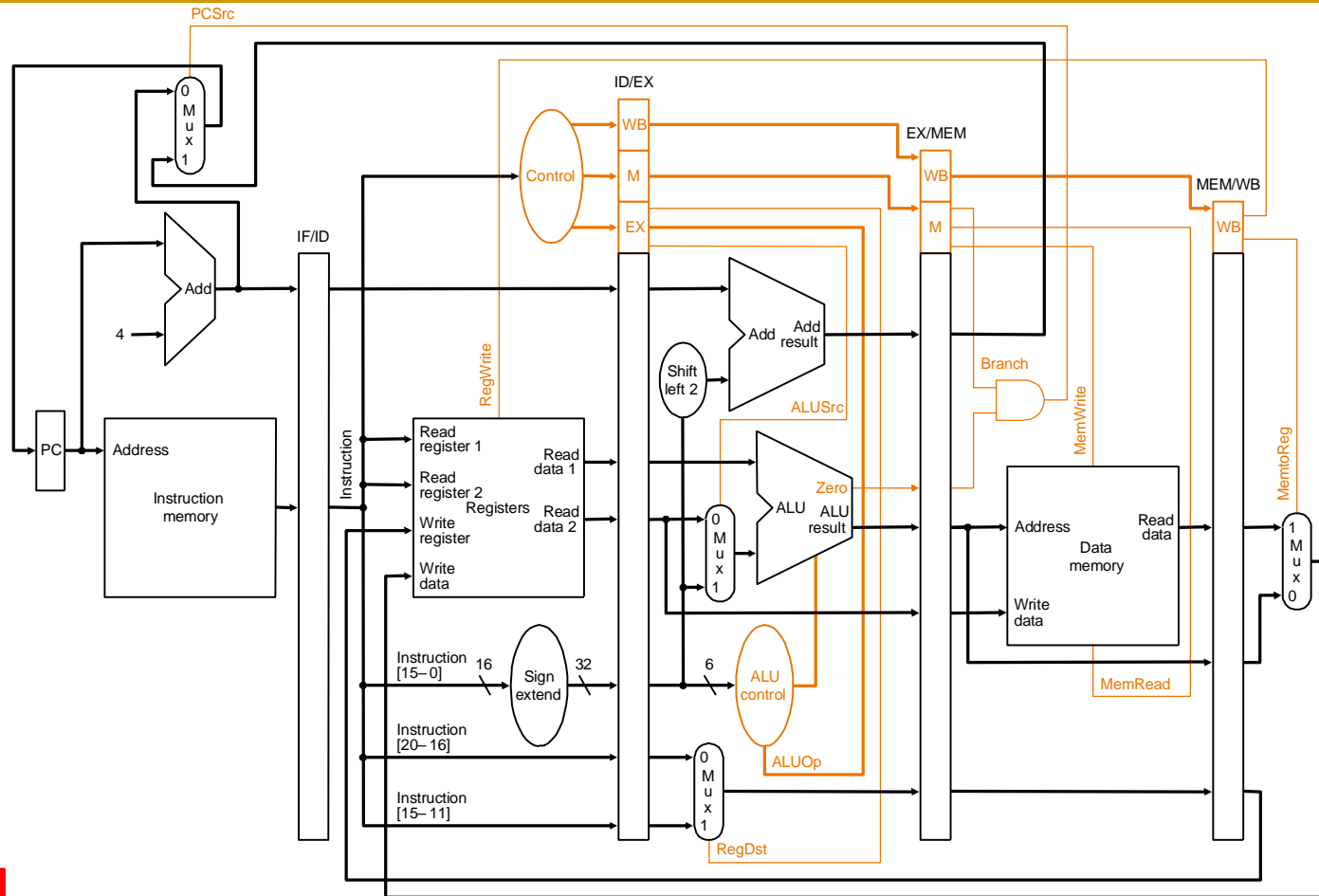
RAW Dependence Handling

- Which one of the following flow dependences lead to conflicts in the 5-stage pipeline?





How to Implement Stalling

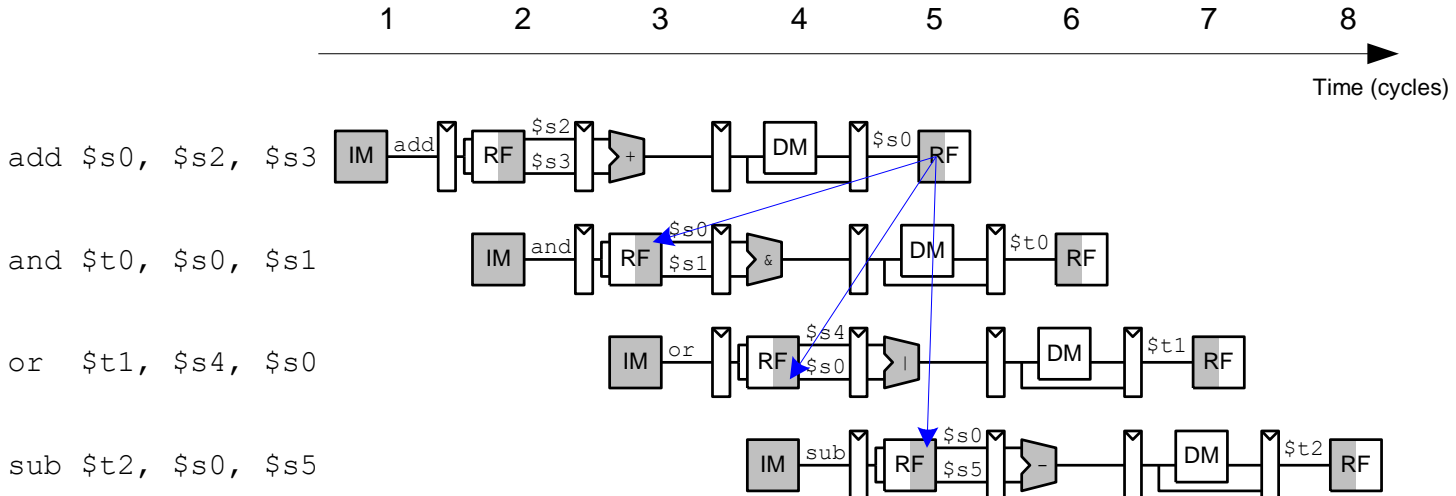


■ Stall

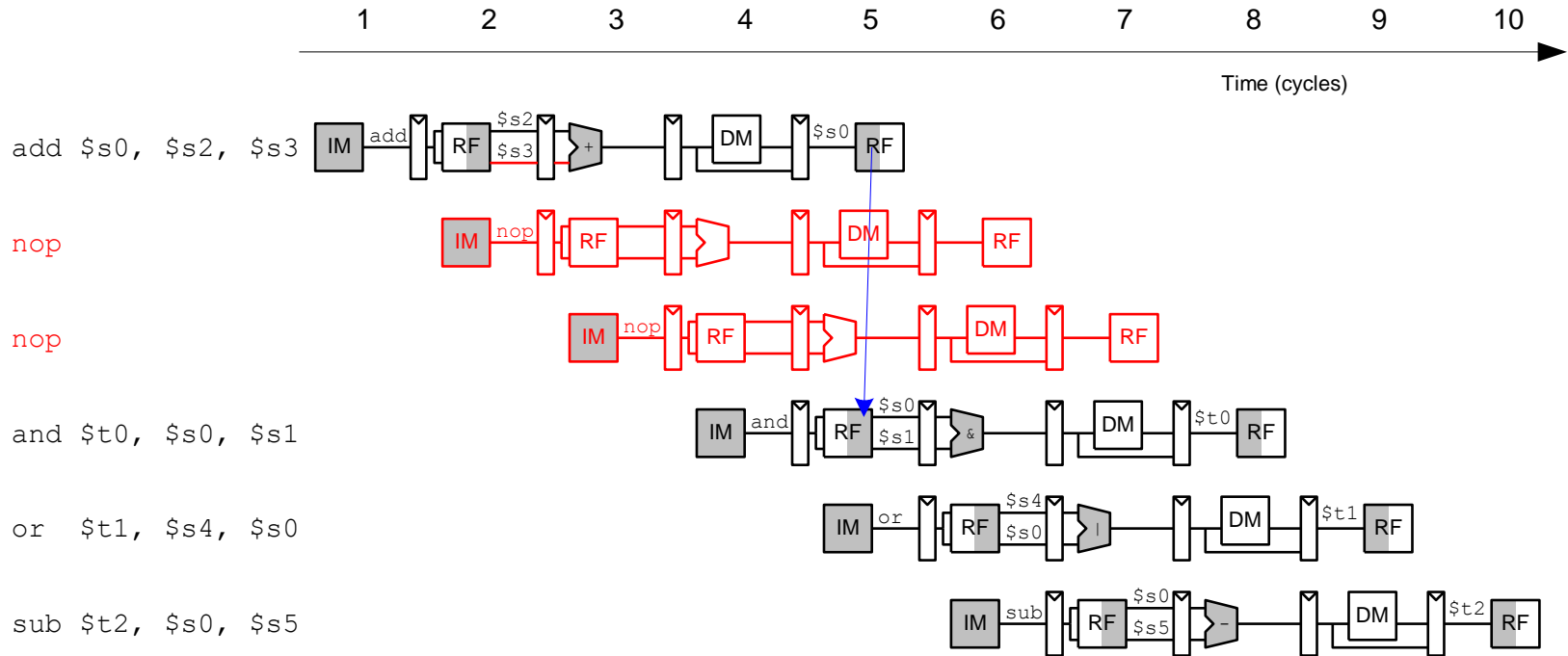
- ❑ disable **PC** and **IF/ID** latching; ensure stalled instruction stays in its stage
- ❑ Insert **"invalid"** instructions/nops into the stage following the stalled one (called **"bubbles"**)

RAW Data Dependence Example

- One instruction writes a register (\$s0) and next instructions read this register => read after write (RAW) dependence.
 - *add* writes into \$s0 in the first half of cycle 5
 - *and* needs \$s0 on cycle 2, obtaining the wrong value
- Only if the pipeline handles data dependences wrong!
- *sub* reads \$s0 in the second half of cycle 5, obtaining the correct value
 - subsequent instructions read the correct value of \$s0



Compile-Time Detection and Elimination

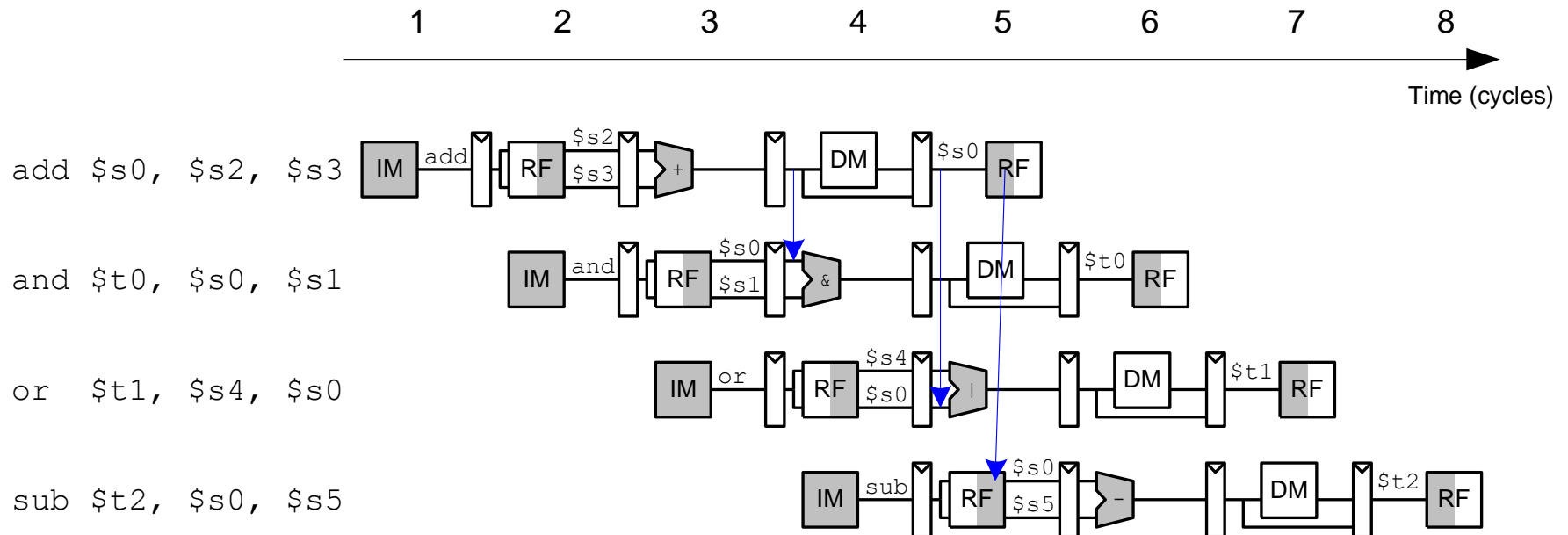


- Insert enough NOPs for the required result to be ready
- Or (if you can) move independent useful instructions up

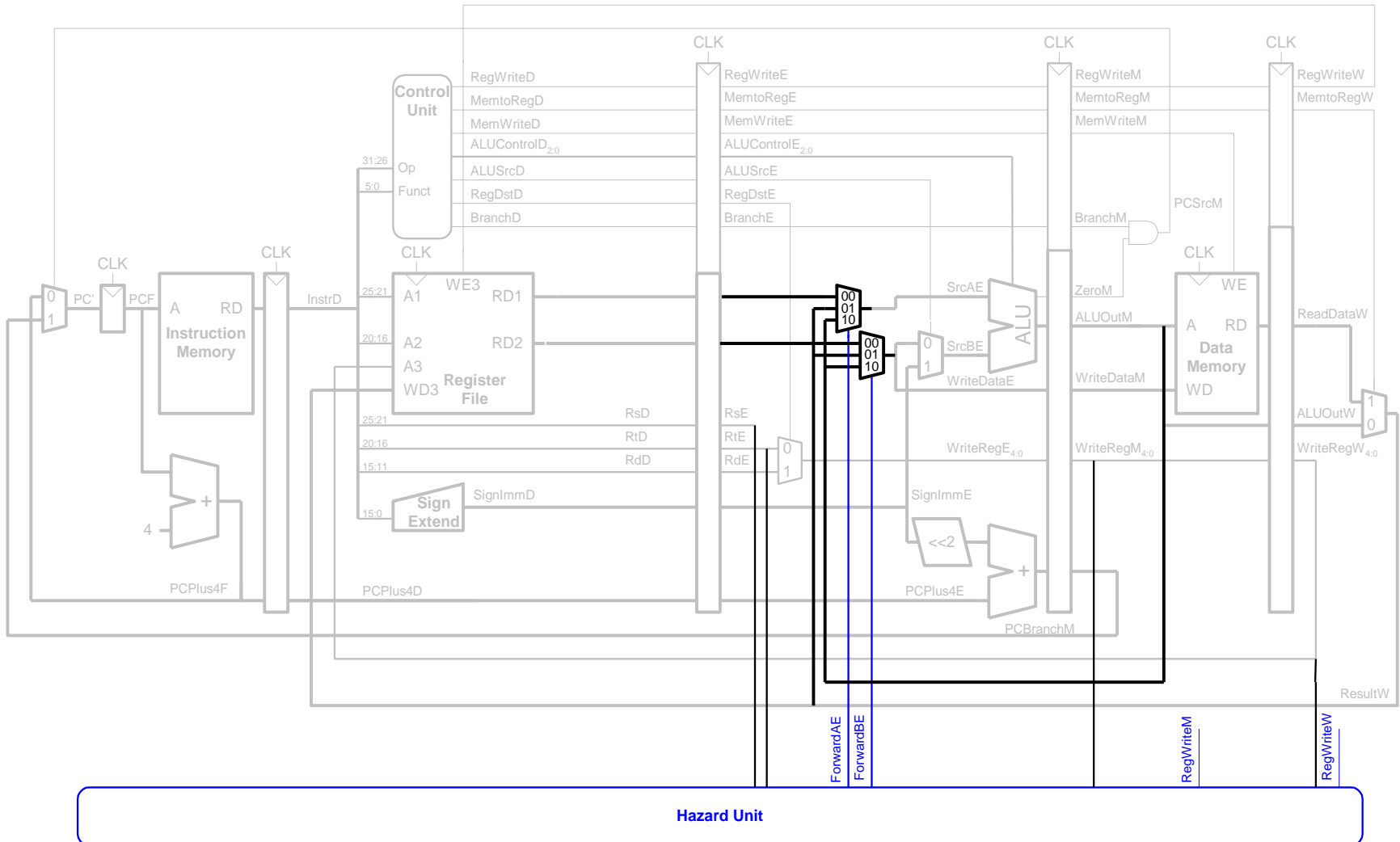
Data Forwarding

- Also called **Data Bypassing**
- We have already seen the basic idea before
- **Forward the result value to the dependent instruction as soon as the value is available**
- **Remember dataflow?**
 - Data value supplied to dependent instruction as soon as it is available
 - Instruction executes when all its operands are available
- **Data forwarding brings a pipeline closer to data flow execution principles**

Data Forwarding



Data Forwarding



Data Forwarding

- **Forward to Execute stage from either:**
 - Memory stage or
 - Writeback stage
- **When should we forward from one either Memory or Writeback stage?**
 - If that stage will write a destination register and the destination register matches the source register.
 - If both the Memory and Writeback stages contain matching destination registers, the Memory stage should have priority, because it contains the more recently executed instruction.

Data Forwarding

- **Forward to Execute stage from either:**

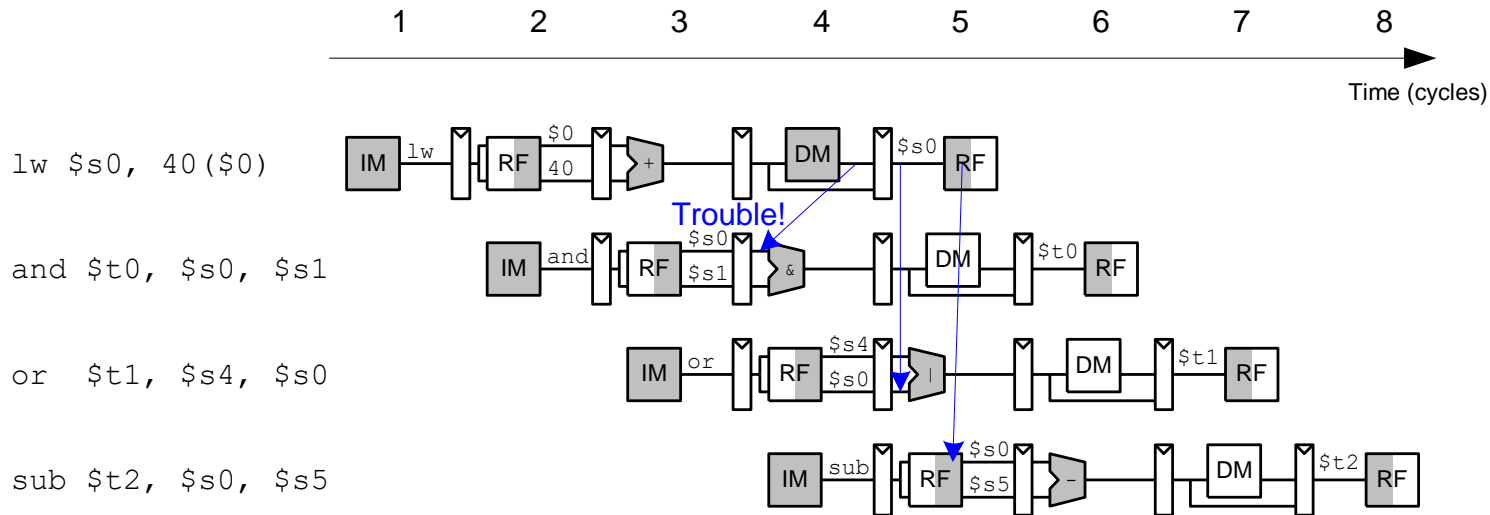
- Memory stage or
- Writeback stage

- **Forwarding logic for *ForwardAE* (pseudo code):**

```
if      ((rsE != 0) AND (rsE == WriteRegM) AND RegWriteM) then
    ForwardAE = 10  # forward from Memory stage
else if ((rsE != 0) AND (rsE == WriteRegW) AND RegWriteW) then
    ForwardAE = 01  # forward from Writeback stage
else
    ForwardAE = 00  # no forwarding
```

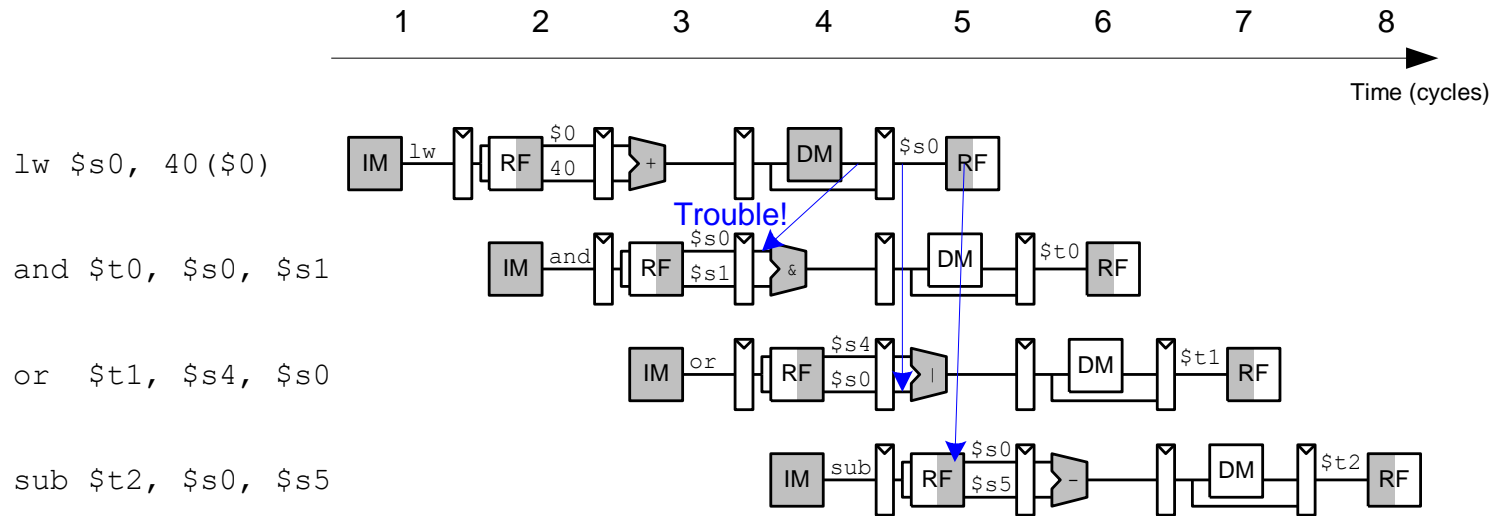
- **Forwarding logic for *ForwardBE* same, but replace *rsE* with *rtE***

Stalling



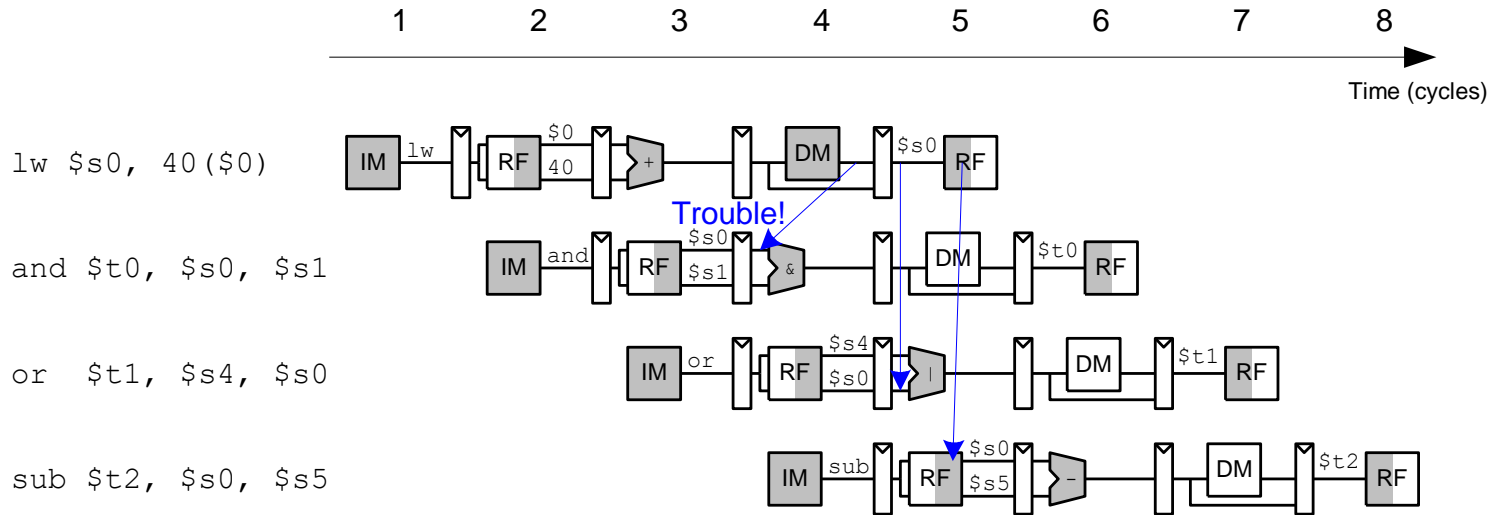
- Forwarding is sufficient to resolve RAW data dependences
- *but ...*
- *There are cases when forwarding is not possible due to pipeline design and instruction latencies*

Stalling



The **lw** instruction *does not finish* reading data until the end of the Memory stage, so its result *cannot be forwarded* to the Execute stage of the next instruction.

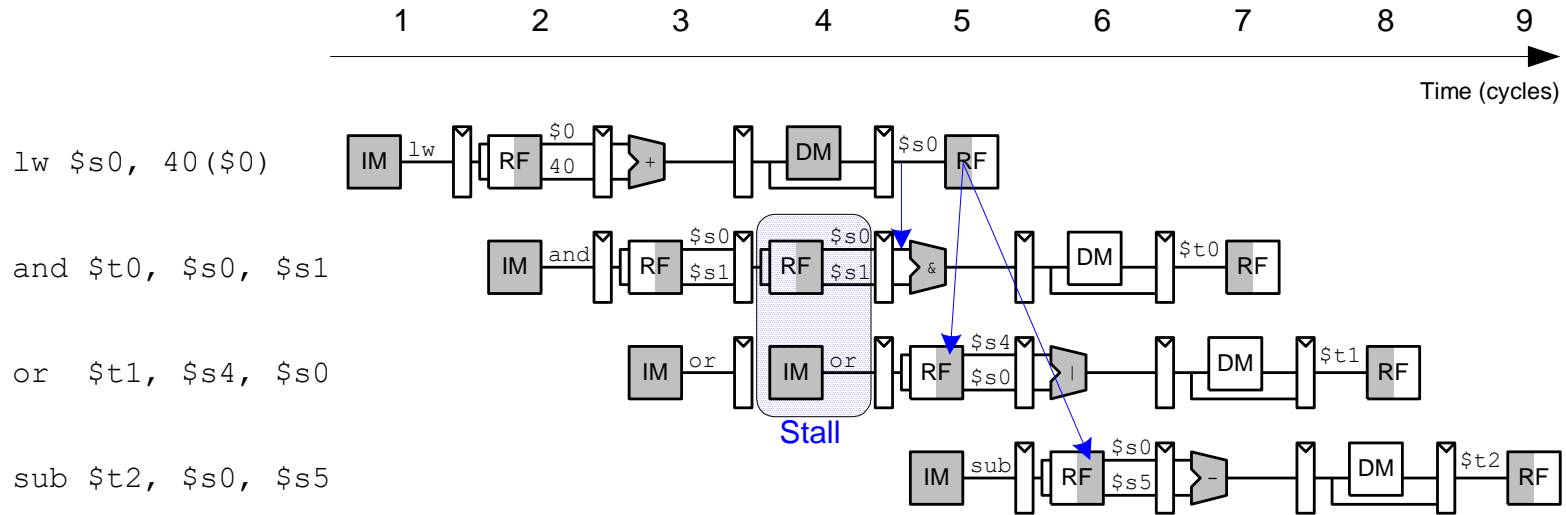
Stalling



The **lw** instruction has *a two-cycle latency*, therefore a dependent instruction cannot use its result until two cycles later.

The **lw** instruction receives data from memory at the end of cycle 4. But the **and** instruction needs that data as a source operand at the beginning of cycle 4. *There is no way to supply the data with forwarding.*

Stalling



Stalling Hardware

■ Stalls are supported by:

- adding enable inputs (EN) to the Fetch and Decode pipeline registers
- and a synchronous reset/clear (CLR) input to the Execute pipeline register
 - or an INV bit associated with each pipeline register

■ When a lw stall occurs

- StallD and StallF are asserted to force the Decode and Fetch stage pipeline registers to hold their old values.
- FlushE is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble

Fine-Grained Multithreading

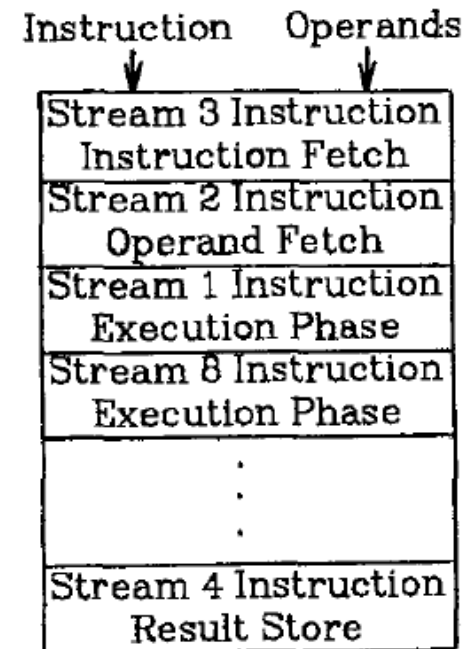
How to Handle Data Dependences

- Anti and output dependences are easier to handle
 - write to the destination in one stage and in program order
- Flow dependences are more interesting
- Five fundamental ways of handling flow dependences
 - Detect and wait until value is available in register file
 - Detect and forward/bypass data to dependent instruction
 - Detect and eliminate the dependence at the software level
 - No need for the hardware to detect dependence
 - Predict the needed value(s), execute “speculatively”, and verify
 - Do something else (fine-grained multithreading)
 - No need to detect

Fine-Grained Multithreading

- Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.
 - By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
 - Branch/instruction resolution latency overlapped with execution of other threads' instructions

- + No logic needed for handling control and data dependences within a thread
- Single thread performance suffers
- Extra logic for keeping thread contexts
- Does not overlap latency if not enough threads to cover the whole pipeline



Fine-Grained Multithreading (II)

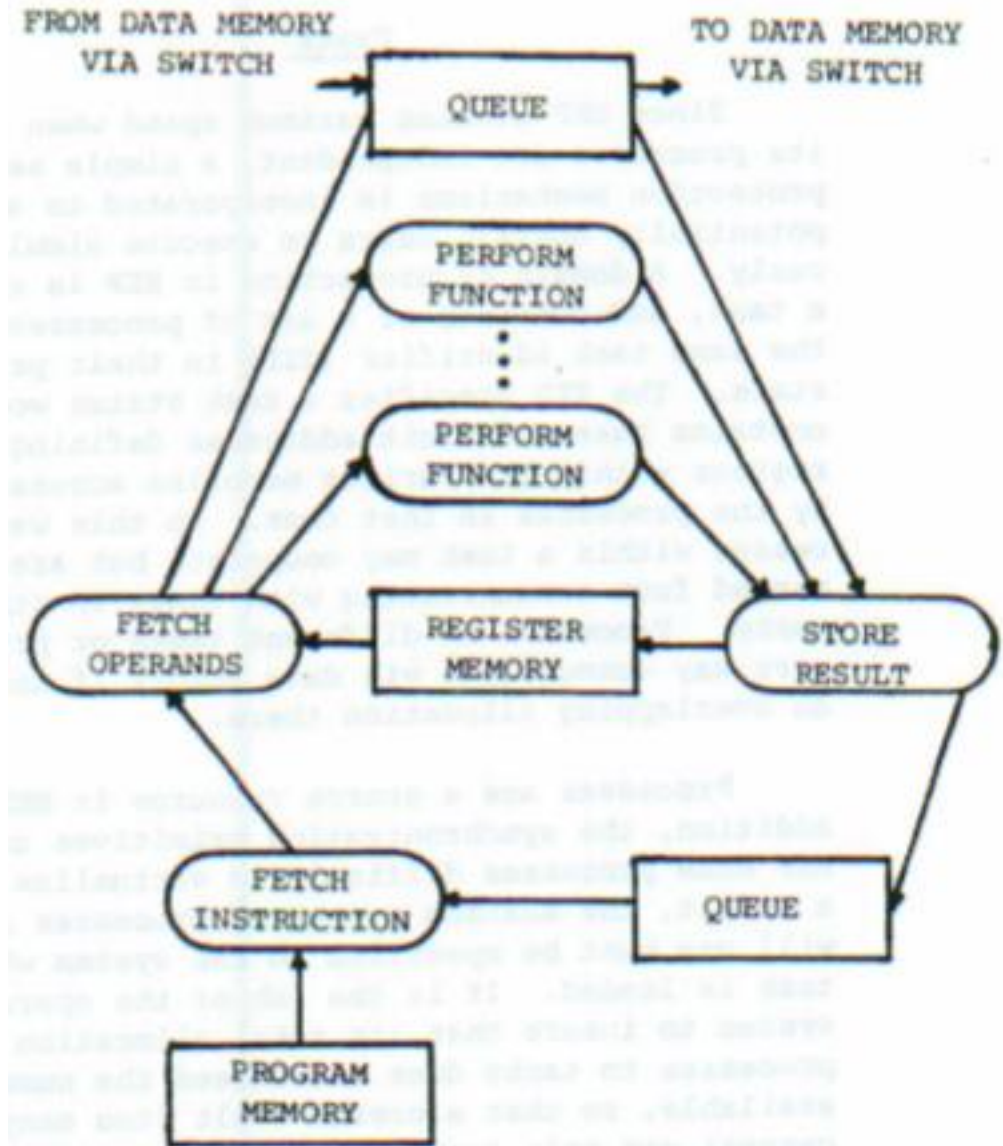
- Idea: Switch to another thread every cycle such that no two instructions from a thread are in the pipeline concurrently
- Tolerates the control and data dependency latencies by overlapping the latency with useful work from other threads
- Improves pipeline utilization by taking advantage of multiple threads
- Thornton, “Parallel Operation in the Control Data 6600,” AFIPS 1964.
- Smith, “A pipelined, shared resource MIMD computer,” ICPP 1978.

Fine-Grained Multithreading: History

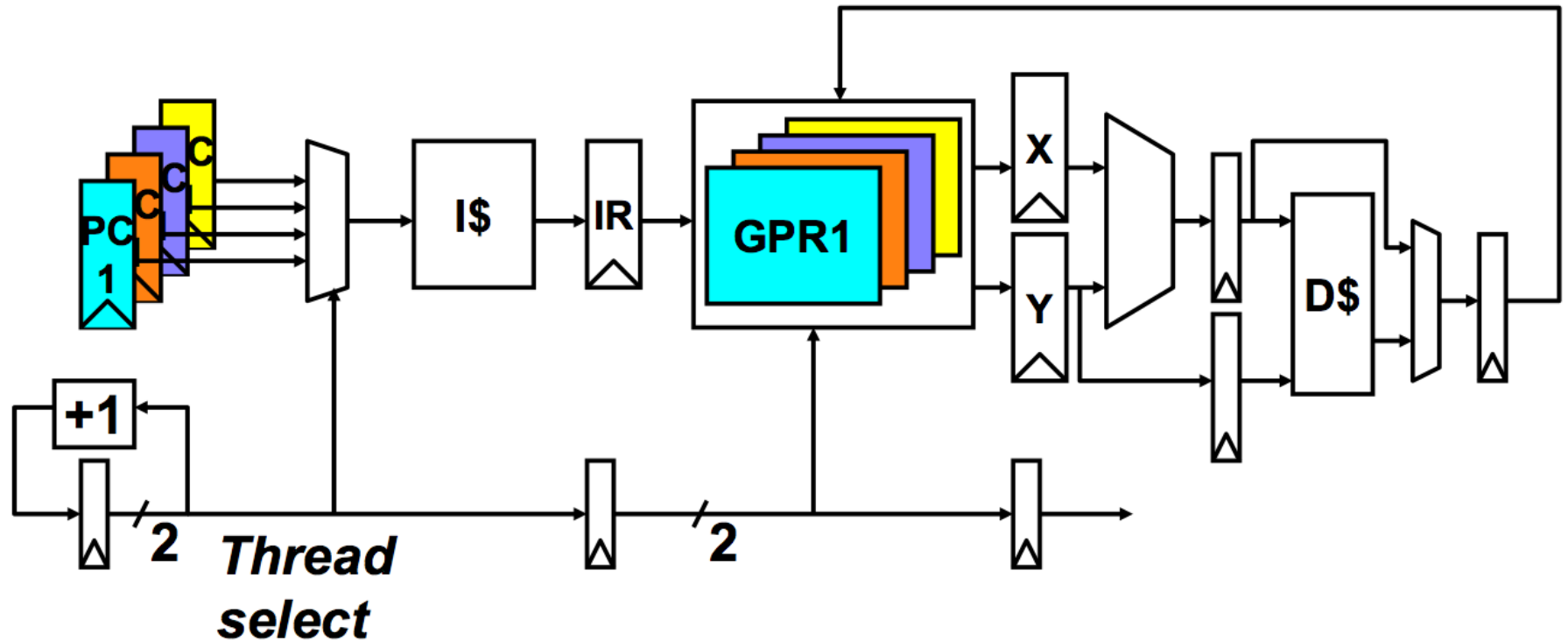
- CDC 6600's peripheral processing unit is fine-grained multithreaded
 - Thornton, “[Parallel Operation in the Control Data 6600](#),” AFIPS 1964.
 - Processor executes a different I/O thread every cycle
 - An operation from the same thread is executed every 10 cycles
- Denelcor HEP (Heterogeneous Element Processor)
 - Smith, “[A pipelined, shared resource MIMD computer](#),” ICPP 1978.
 - 120 threads/processor
 - available queue vs. unavailable (waiting) queue for threads
 - each thread can have only 1 instruction in the processor pipeline; each thread independent
 - to each thread, processor looks like a non-pipelined machine
 - system throughput vs. single thread performance tradeoff

Fine-Grained Multithreading in HEP

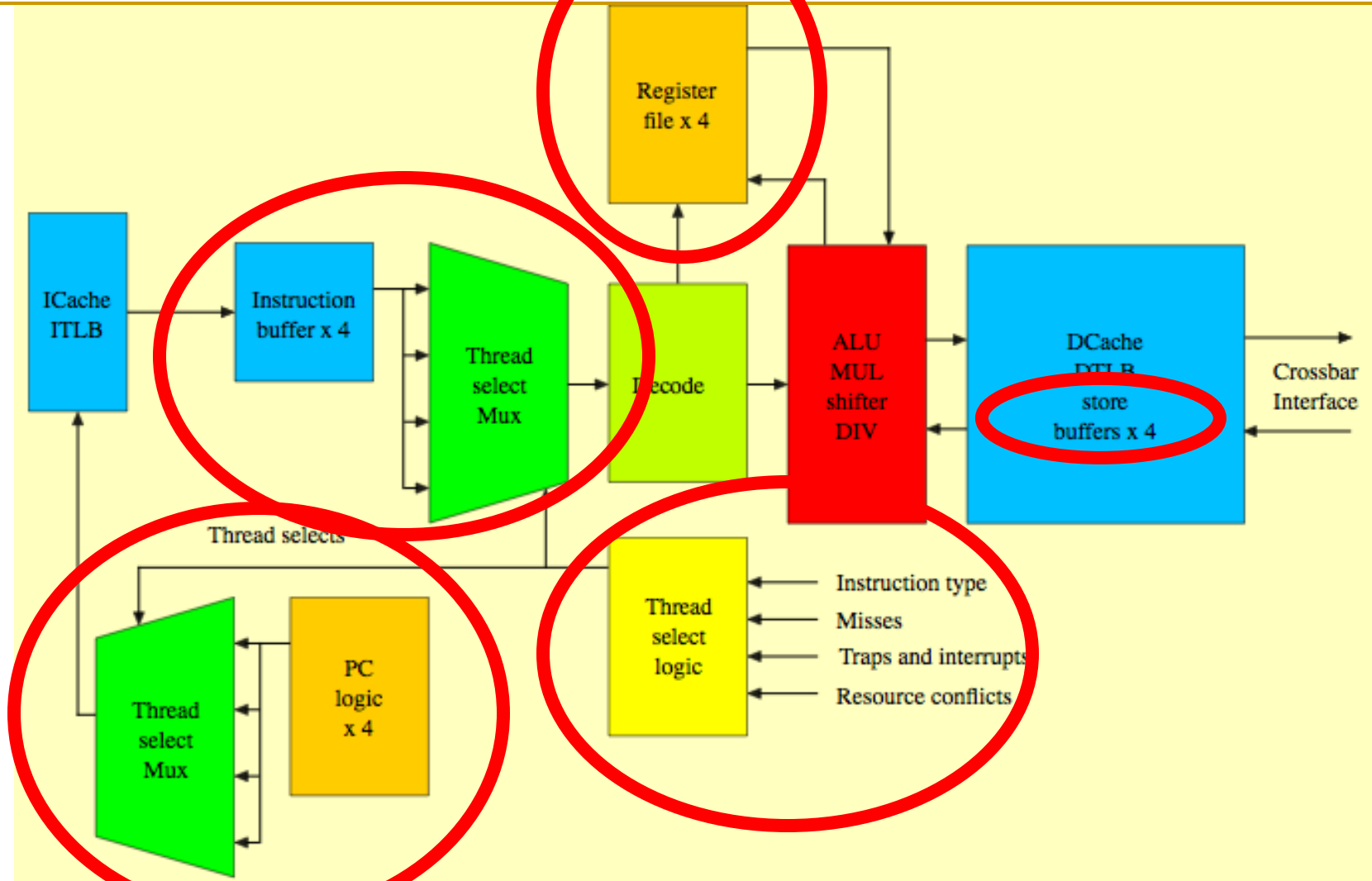
- Cycle time: 100ns
- 8 stages → 800 ns to complete an instruction
 - assuming no memory access
- No control and data dependency checking



Multithreaded Pipeline Example



Sun Niagara Multithreaded Pipeline



Kongetira et al., "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro 2005.

Fine-grained Multithreading

■ Advantages

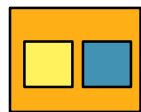
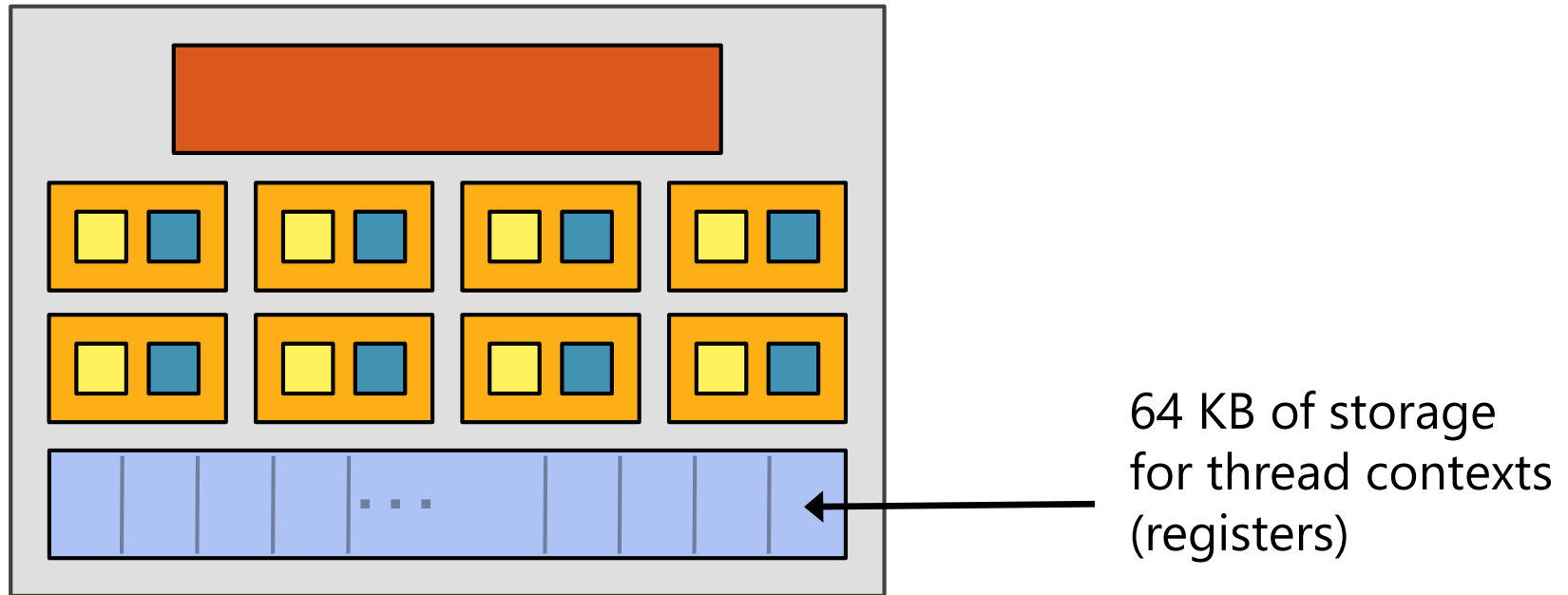
- + No need for dependency checking between instructions
(only one instruction in pipeline from a single thread)
- + No need for branch prediction logic
- + Otherwise-bubble cycles used for executing useful instructions from different threads
- + Improved system throughput, latency tolerance, utilization

■ Disadvantages

- Extra hardware complexity: multiple hardware contexts (PCs, register files, ...), thread selection logic
- Reduced single thread performance (one instruction fetched every N cycles from the same thread)
- Resource contention between threads in caches and memory
- Some dependency checking logic *between* threads remains (load/store)

Modern GPUs Are FGMT Machines

NVIDIA GeForce GTX 285 “core”



= data-parallel (SIMD) func. unit,
control shared across 8 units



= multiply-add



= multiply

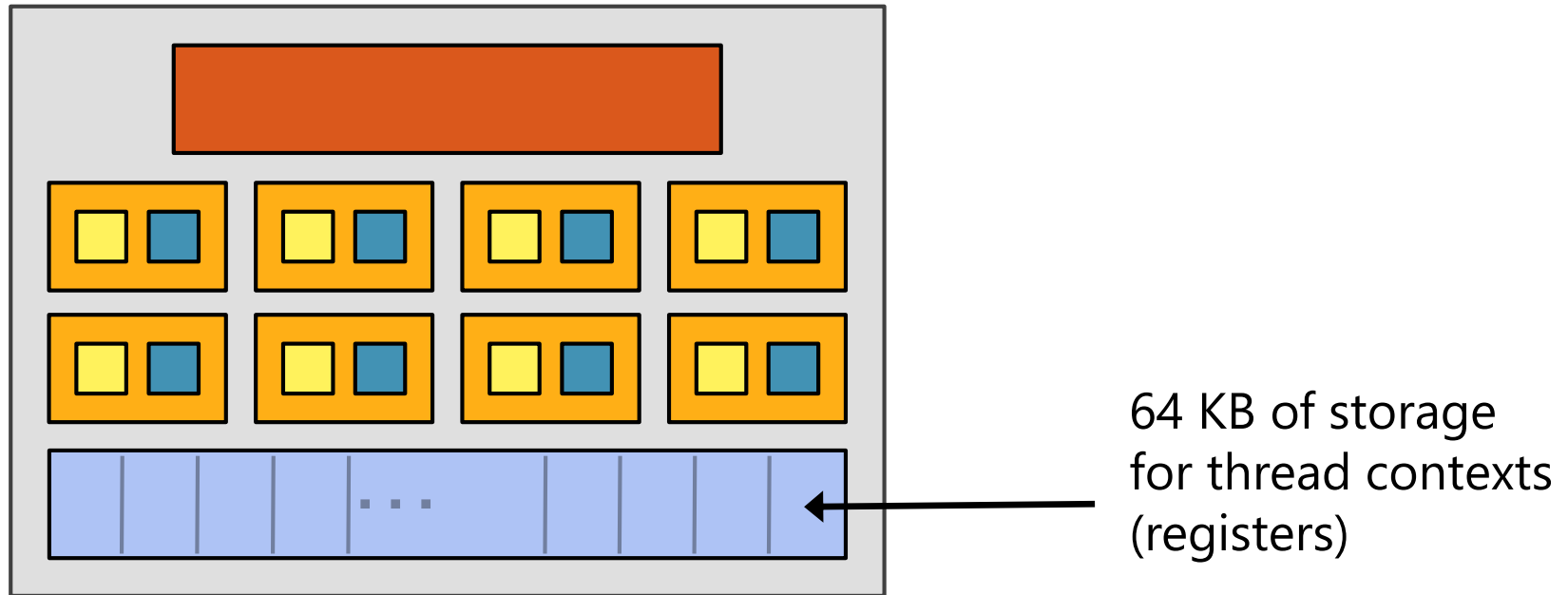


= instruction stream decode



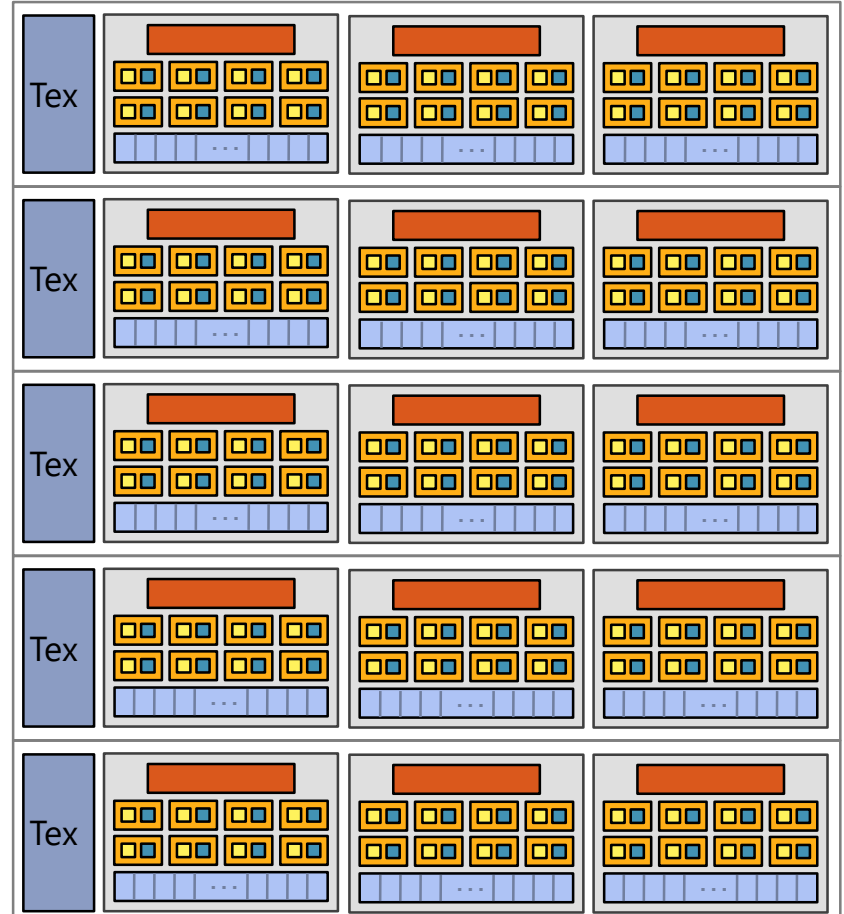
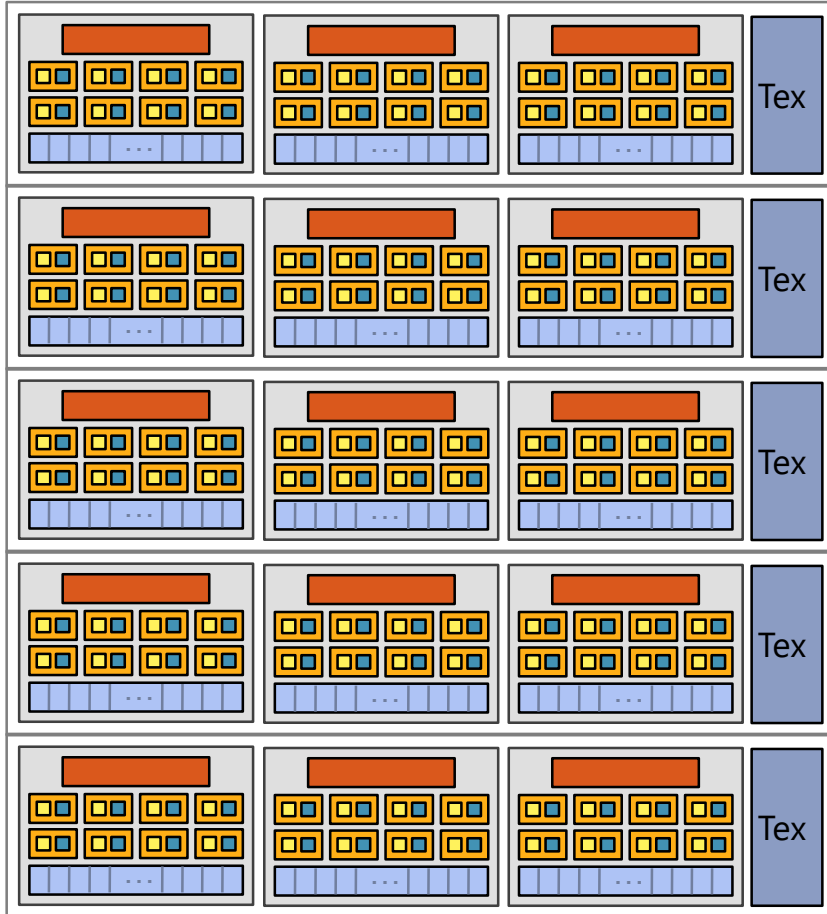
= execution context storage

NVIDIA GeForce GTX 285 “core”



- Groups of 32 **threads** share instruction stream (each group is a Warp): they execute the same instruction on different data
- **Up to 32 warps are interleaved in an FGMT manner**
- Up to 1024 thread contexts can be stored

NVIDIA GeForce GTX 285



30 cores on the GTX 285: 30,720 threads

End of Fine-Grained Multithreading